Beyond Finite Memory With Networks Of Spiking Neurons

Ashley John Spencer Mills

April 8, 2004

Abstract

Feed-forward networks of spiking neurons with recurrent connections are trained on temporal symbolic sequences to induce persistent stable structures entailing their application to the emulation of specific target Moore machines. Target Moore machines that enable a trained network to demonstrate an ability to go beyond finite memory are shown inducible on training sets constructed from several concatenated short temporal symbolic sequences consistent with their operation. This work was performed in collaboration with Dr. Peter Tinŏ, was initiated in July 2003, and funded by The Nuffield Science Foundation.

THE UNIVERSITY
OF BIRMINGHAM

BSc in Computer Science – School of Computer Science – Supervised by Dr. Peter Tiňo

Contents

1 Introduction

This section describes the progression from artificial neural networks to spiking neural networks, motivates their use, and puts the research in context with respect to related research performed by others.

In [18], neural network models are classified into three generations, the following three paragraphs recapitulate on that.

The first generation of artificial neural network models were based on *McCulloch-Pitts* [21] threshold-gates and were only capable of computing on digital inputs, producing digital outputs. It was shown theoretically that networks of these neurons were capable of simulating arbitrary boolean threshold functions. For more information on the limitations of these types of networks, see [22].

The second generation of artificial neural network models applied activation functions at the neuron level (typically sigmoidal) to a sum of incoming analog inputs producing a continuously valued output at each neuron. It was shown theoretically, that in addition to being able to simulate any boolean threshold function, networks of these neurons were capable of approximating, to arbitrary precision, any well behaved function [10]. Second generation networks were based on the idea that information is propagated by means of a rate code; where the firing rate of a neuron is used to transmit information, this was consistent with certain observed biological results. In many second generation network models the sigmoidal activation of a neuron represented its firing rate. Second generation networks were considered to be more biologically plausible models of real neural networks than first generation networks.

Experimental results from neurobiology showed that biological organisms employ neuronal processing for critical functions over temporal windows so short that the amount of information transmittable using a rate code is unusable, indicating that in these circumstances information is being encoded in the precise timing of individual spikes. It became widely accepted that the timing of individual spikes are used to transfer information in many biological systems (See [2] and references therein). There was consequently an emergence of interest in the artificial modeling of networks of spiking neurons; the third generation of neural network models, and the primary paradigm considered in this research. It has been shown theoretically that third generation networks are universal approximators for digital and analog functions [14].

For an extensively referenced introductory article on spiking neuron networks, that assumes no prior knowledge, see [18]. For a good overview of the most active research areas, see [19]. For a formal definition of a spiking neuron network, see [16].

Finite state machines (FSM) are good models for computations on time series, as well as obtaining significance in theoretical computer science and linguistics. Since networks of spiking neurons deal exclusively with temporal information, it is interesting to assess their ability to perform computations on time series, a good way to do this is by asking whether they can emulate the operation of arbitrary FSM, devise a training mechanism that attempts to teach an RSN to act like a particular RSN, and then study the learning process. Much research has been done using second generation networks with recurrent connections, referred to here as classical recurrent networks (CRN), in assessing their capacity to induce FSM [4, 6, 3, 28, 29, 31]. Significantly less research has been done into the same problem using third generation networks (Feed Forward Spiking Networks, FFSN) with recurrent connections, referred to here as RSN (Recurrent Spiking Networks).

In [27], FFSN are empirically shown capable of emulating arbitrary FSM taken from the sub-class of Definite Memory Machines (DMM) whose output relies on at most three previous inputs and the current input. The previous inputs are stored in the synapses of a delaying network, which employs a biologically inspired model [12], defined by recursive equations governing the membrane potential of neurons, to present delayed copies of the current input at finite intervals in the future within a finite temporal window, to a trainable FFSN. According to [27] it has been shown in [15] that in theory such networks can implement all time invariant filters with fading memory, where a filter is a function that produces a vector of output firing times given an vector of input firing times, time invariance implies that a temporal translation of the inputs obtains an equivalent temporal translation at the outputs, maintaining invariance in the relative firing times of inputs and outputs, and fading memory means that the output for any given input can depend only on a finite number of preceding inputs.

The aim of this research is to investigate the ability of networks of spiking neurons with recurrent connections (recurrent spiking networks, RSN) to induce persistent structures which obtain infinite memory; in the sense that the output for any given input, depends on *every* previous input since the beginning of a given simulation. This is achieved using a network architecture with recurrent connections, which allows the operation of a Moore machine (MM), a special type of FSM, to be emulated. The infinite memory is achieved by virtue of the recurrent connections providing complete knowledge of the states in the Moore machine that would occur if the transitions on all previous inputs beginning at the start state were followed, such that given the current input, the correct next state will be predicted via the firing times of the recurrent connections, and the output associated with the next state will be predicted over the output neurons. Training is performed on input sequences derived from a target MM, M, the aim is to train an RSN with the Spikeprop local gradient descent learning rule [25] to predict the correct output sequences for arbitrary input sequences, where the input symbols allowed in the input sequences are constrained according to those inputs actually used by M. Although strictly it is the RSN that obtains infinite memory, or in other words that goes beyond finite memory, it is convenient to say that an MM, M, goes beyond finite memory if, when induced by an RSN, allows the RSN to demonstrate infinite memory through its emulation of M. For an example of an MM that goes beyond finite memory see §4.2 (page-22).

§1 introduces the spike response model, which is the core component of computation with networks of spiking neurons, motivates the use of spikes, formally describes networks of spiking neurons, and describes how to perform supervised gradient descent learning with networks of spiking neurons. §3 describes how various types of spiking neuron network are simulated by computer in this research, and how learning mechanisms are implemented. §4 describes how to induce the structure of a target MM in the weights of an RSN. §5 describes how to extract induced structures from trained RSN so that their architectures can be compared with those of target MM. §6 describes some empirical experiments which show the induction of FSM that go beyond finite memory. §7 evaluates the research semi-retrospectively. §8 discusses possible induction mechanisms . §9 summarises and concludes.

Appendix-B takes a look at the design and implementation of the simulation software from the perspective of a software engineer. Appendix-A contains formal algorithms for the computer simulations discussed in §3. Appendix-C presents the parameters used for the experiments of §6 in detail.

2 The Spike Response Model

2.1 Motivation and Theory

Although this paper is principally concerned with the ability of networks of spiking neurons to encode information in the timing of individual spike transmission, this is by no means the only paradigm for computation with networks of spiking neurons. There exists empirical evidence to suggest that real biological networks of spiking neurons may use rate codes over relatively longer time scales, in addition to using spike codes over,respectively , relatively shorter time scales, to perform computation. It was shown in [17] that networks of spiking neurons can simulate arbitrary feed forward networks of sigmoidal networks, and the possibility that biological systems can employ an analogous computational mechanism, is argued in the positive.

The spiking neuron network model used in this paper utilises the relative timing of individual spikes to propagate information about inputs through a constructed network to its output units. The firing times of the output units are used in a back-propagation-like learning rule to adjust the network so that its outputs for a given input approximate more readily the target function. This report employs so-called leaky integrate-and-fire spiking neurons arranged in fully connected feed forward networks, with recurrent connections. Figure-1 shows a diagram of a biological synapse and an electron microphoto of similar structure, Figure-2 shows the basic components of a neuron.

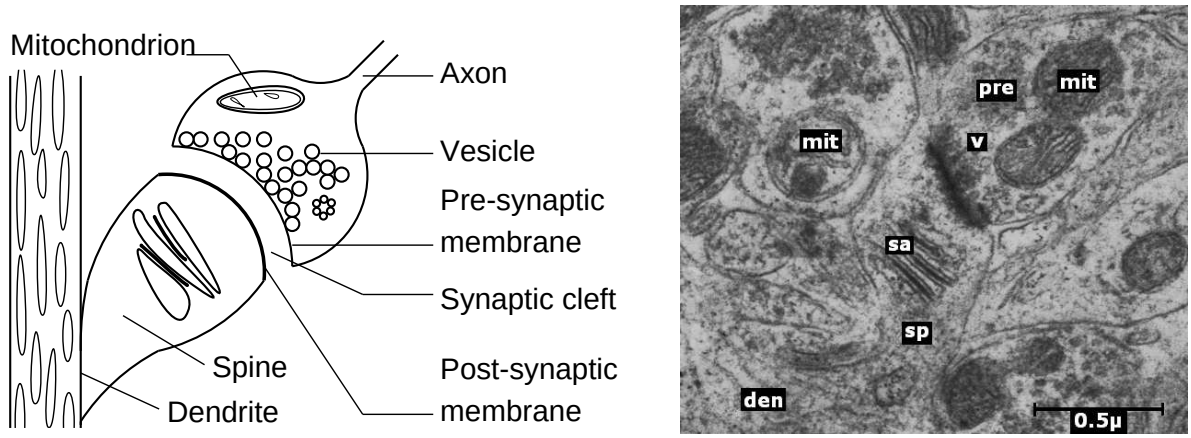


Figure 1: **Left:** Basic components of a biological synapse, as labeled. **Right:** Electron microphoto of a synapse connecting a presynaptic knob *(pre)* to the spine *(sp)*, of an apical dendrite *(den)* of a cortical pyramidal cell. Other labeled elements are: mitochondria *(mit)*, accumulated vesicles *(v)*, and spine apparatus *(sa)*. The synaptic cleft is located at the dark band near *(v)*. Electron microphoto credit: [11] (Modified, and used with permission courtesy of Blackwell Publishing Oxford).

In biological systems, when the membrane potential of a post-synaptic knob exceeds some threshold, voltage gated ion channels open and a rapid depolarisation of the postsynaptic soma is actuated, causing an electrotonus and consequently the flow of current down the axon of the postsynaptic neuron to synapses attached to the soma, dendrites, and initial axon segments of other neurons. Upon reaching each synaptic knob terminating these conductive paths – where the following description applies to a single synapse – the consequent electrotonus brings about liberation of neurotransmitter into the synaptic cleft through the fusing of synaptic vesicles with the presynaptic membrane. This neurotransmitter diffuses across the synaptic cleft and is taken up by postsynaptic receptors bringing about an increase or decrease in the membrane potential at the postsynaptic site thus potentially leading to further propagation of impulses in the manner just described. After firing, the membrane potential of a neuron must sufficiently repolarise before it can fire again. Following the firing of a neuron, there is a short duration called the absolute refractory period within which the neuron absolutely cannot fire again, irrespective of the size of any stimulus, this is followed by a longer period known as the relative-refractory period, within which the neuron *can* fire, but the amount of stimulus required to initiate firing is greater than normal, following

this the amount of stimulus required gradually decreases through time until it reaches some constant value, in this state the neuron is said to obtain a resting potential. Reference to the refractoriness of a neuron is therefore a reference to the functions that describe its absolute and relative refractory behaviour.

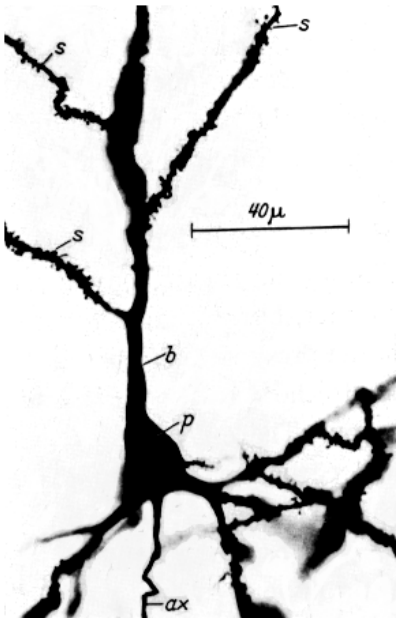


Figure 2: Golgi preparation of a neuron from cat cerebral cortex with spines *(s)* shown on apical and basal dendrites, but not on the soma *(p)*, axon *(ax)*, or dendritic stumps *(b)*. Description credit [9], image credit [30], used with permission courtesy of Oxford University Press.

The precise dynamics of a particular synapse are involved in modulating the effect that neurotransmitter released from the presynaptic knob has on the postsynaptic membrane. Individual impinging synaptic connections can be discordant with respect to their contribution to the overall alteration of the membrane potential at a postsynaptic site; the neurotransmitter released can have either an inhibitory or an excitatory effect on the membrane potential. The magnitude of the effect is a function of several things: the neurotransmitters released, the number of synaptic connections potentiating the electrotonus, the number of receptors for these neurotransmitters at the postsynaptic site, the firing history of the neuron, and many other dynamic influences. Extensive, and extremely detailed research has been done into the dynamics of synapses and neurones (for e.g [9]). Some computational neuroscientists have even gone so far as to suggest that computation in neurons is at a level lower than the neuron, and that *hot spots*; points on the dendritic trees of neurons with voltage gated channels, can communicate information via dendritic spikes; actuated at a particular hot spot by the membrane potential exceeding some threshold, such that, in effect, a single neuron may have the ability to act like an entire network of sigmoidal units, (See Remark 4 in [17], and references therein).

In light of this complexity it is desirable to simplify and abstract to a neuron model consistent with the majority of biological findings, but simple enough to analyse theoretically. A candidate for this is the spike response model, which describes the membrane potential dynamics of a neuron relative to the firing of presynaptic neurons and itself. This is the model currently most often employed in the artificial simulation of spiking neurons, and in the analysis of their computational complexity. A very general form of this is described by Equation-1.

$$u_j(t) = \sum_{t_j^a \in \mathcal{F}_j} \eta_j(t - t_j^a) + \sum_{i \in \Gamma_j} \sum_{t_i^a \in \mathcal{F}_i} w_{ij} \cdot \epsilon_{ij}(t - t_i^a) \tag{1}$$

Where $u_j(t)$ is the membrane potential of neuron j at time t. The left hand term sums the responses from a kernel η_j, modeling the refractoriness of the neuron j, when fed the differences between the current time t and all previous firing times $\mathcal{F}_j$ of neuron j. The right hand term sums the synaptic efficacy w_{ij} weighted contributions to the membrane potential of the postsynaptic neuron j over all the previous firing times $t_i^a \in \mathcal{F}_i$ of all presynaptic neurons $i \in \Gamma_j$, where ϵ_{ij} is a kernel that models the change in membrane potential that a particular firing instance t_i^a of a particular presynaptic neuron t_i educes in the postsynaptic neuron t_j.

The indexes on the kernels η, and ϵ allow for arbitrary choosing of kernels for respectively, individual neurons i and individual synaptic connections ij. Typically the kernel η_j might be something like Equation-2.

$$\eta_j(s) = -\eta_0 \cdot exp\left(-\frac{s - \delta^{abs}}{\tau}\right) \cdot \mathcal{H}(s - \delta^{abs}) - K \cdot \mathcal{H}(s) \cdot \mathcal{H}(\delta^{abs} - s) \tag{2}$$

Where s is the time since firing of neuron j, $-\eta_0$ scales the refractoriness of the function, δ^{abs} is the absolute refractory period within which the neuron cannot fire again, $\mathcal{H}$ is a Heaviside step function for

which is 1 for $s > 0$ otherwise 0, and $K \to \infty$ is a constant ensuring that δ^{abs} is absolutely refractory. This kernel effectively realises a dynamic threshold dependent on the previous firing times of neuron j.

The kernel ϵ_{ij} models the effect that presynaptic spike-induced neurotransmitter release has on the postsynaptic membrane potential. This contribution may be excitatory or inhibitory. The linear superposition of all such contributions models the postsynaptic membrane potential at time t. There are many choices for the kernel ϵ, but it is desirable to use something biologically inspired.

$\mathcal{F}_j$ is the set of all firing times of neuron j at a time less than t. Formally it can be described by Equation-3.

$$\mathcal{F}_j = \{t | (u_j(t) = \vartheta) \wedge u'_j(t) > 0\} \tag{3}$$

Where ϑ is a threshold, and $u'_j(t)$ is the derivative of the membrane potential at time t ensuring that a firing time is defined as the membrane potential exceeding the threshold ϑ *from below*. This set of firing times $\mathcal{F}_j$ is called a Spiketrain. For a more thorough analysis of the mathematics presented here, see Chapter 1 of [19], from which the equations presented above were taken, and the descriptions based.

In this research Equation-1 is constrained by removing the left hand term of the sum, which removes neuron refractoriness from the model. This simplification is accomplished by constraining the simulation so that each neuron can fire only once within each simulation run, and by assuming that there has been sufficient time since the last firing time t_i^a of each presynaptic neuron $i \in \Gamma_j$ so that any increase in the threshold due to the previous firing times of the presynaptic neurons, that would have been realised by Equation-2, is insignificant enough to be ignored completely. This constraint also ensures that only the last firing time t_i^a of the presynaptic neurons $i \in \Gamma_j$ need be considered when calculating the right hand term of the addition in Equation-1, because of the constraint that each neuron may fire only once within each simulation run. After imposing these constraints, Equation-4 is obtained:

$$u_j(t) = \sum_{i \in \Gamma_j} w_{ij} \cdot \epsilon_{ij}(t - t_i^a) \tag{4}$$

Only two types of response kernel ϵ will be considered: the first having an excitatory affect on the postsynaptic membrane potential, and the second having an inhibitory affect. These types of excitatory and inhibitory affectors are referred to as respectively, EPSP(Excitatory Post Synaptic Potential) and IPSP(Inhibitory Post Synaptic Potential). Furthermore, the simple function shown in Equation-5 is used to model EPSP, and its reciprocal is used to model IPSP. Collectively these stimuli are known as PSPs (Post Synaptic Potentials), owing to the fact that they induce a change in post-synaptic membrane potential. The function used to model PSP used in this research is shown below.

$$\epsilon_{ij}^k(t) = ij_{type}^k \cdot (t/\tau) \cdot exp(1 - (t/\tau)) \cdot \mathcal{H}(t - d_{ij}^k) \tag{5}$$

Where ij_{type}^k is 1 if synapse k between neurons i and j is concerned exclusively with transmitting EPSPs and -1 if it concerned exclusively with transmitting IPSPs. $\mathcal{H}$ is the Heaviside step function which is 1 for $t > 0$, and is otherwise 0, ensuring that the axonal delay d_{ij}^k is enforced. Thus $\epsilon_{ij}^k(t)$ models the contribution of synapse ij^k to the superposition of all PSP affecting the membrane potential of the postsynaptic neuron j at time t. See Figure-3:

Notice that multiple axonal delays were introduced in Equation-5. It is worth briefly considering the motivation for this. In [17] Wolfgang Maass proves that networks of spiking neurons can simulate arbitrary networks of sigmoidal neurons, and so are consequently able to approximate continuous, bounded functions, to arbitrary precision. This is under the constraint that when the PSPs modeled by ϵ_{ijk} from firing presynaptic neurons $i \in \Gamma_j$ impinge on the postsynaptic neuron j, there must be some time t at which all the EPSP and IPSP are in their initially linearly inclining, respectively linearly declining segments, as realised by the response kernel $\epsilon_i j^k(t)$, and they must obtain a simultaneous incidence at the postsynaptic neuron j. This can be achieved by ensuring that all presynaptic neurons fire within a small time interval $[0, \gamma]$, but that constraint is somewhat biologically dubious, therefore it is preferable to remove it.

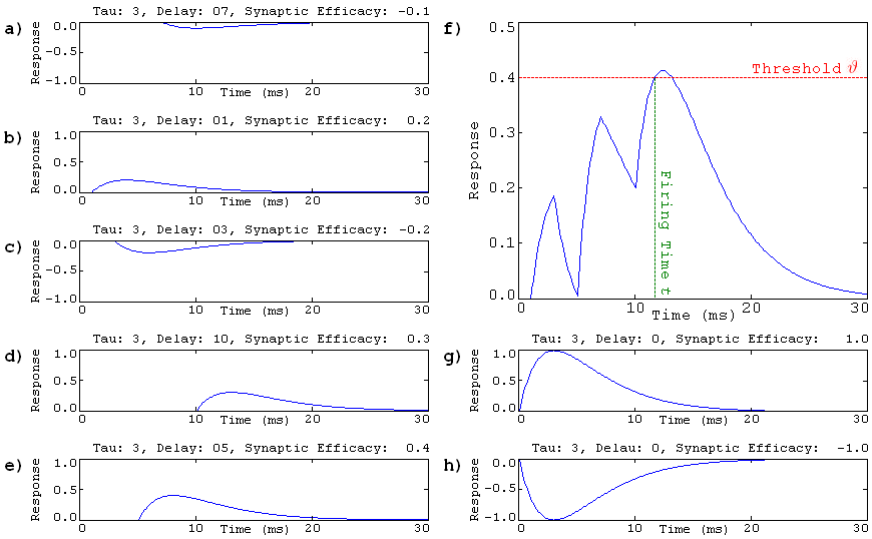


Figure 3: **a-e:** Several realisations of Equation-5, with differing weights (synaptic efficacy), and axonal delays, through time. **f:** The superposition of PSPs a-e through time. Showing the firing time t assuming a threshold ϑ of 0.4. **g:** (EPSP) An excitatory, undelayed realisation of the response function described by Equation-5, with a synaptic weight of 1, through time. **h:** (IPSP) An inhibitory, undelayed realisation of the response function described by Equation-5, with a synaptic weight of 1, through time.

If several delayed overlapping PSP are used, spread out over some time interval, their superposition effectively extends the length of the initial linearly rising or linearly falling segment of the PSP seen at the postsynaptic neuron, such that the possible implementation of a sigmoidal neural net in the spiking neurons is not removed due to theoretical considerations, whilst a longer period $[0, \gamma]$ over which to encode inputs is obtained. This technique was employed in [20] to the emulation of arbitrary Hopfield nets in temporal coding, and in [25] where the Spikeprop learning mechanism, as used in this research, was first introduced and the derivation detailed. In remark 4b of [17] it is suggested that in a practical context, meaningful computation can still be carried out if the linearly rising or falling segments of incoming PSP are spread out over a time interval longer than $[0, \gamma]$, even if they do not overlap, which can happen in simulation due to the attenuation of otherwise overlapping component PSP through weight adjustment.

In much of the computational neuroscience literature referenced in this report, referring to the time between the firing of a presynaptic neuron, and the consequent firing of some affected postsynaptic neuron, there is much talk of synaptic delays, meaning the delay associated with a particular connection between particular pre and postsynaptic neurons, indexed at the synaptic level, in terms of the time it takes for an action potential at the soma of the presynaptic neuron to propagate to the indexed synapse of the presynaptic neuron, diffuse across the synaptic cleft, and propagate along the dendrites of the postsynaptic neuron until its soma is reached. The phrase *synaptic delay*, however, implies a specific reference to exclusively the time it takes for neurotransmitter to diffuse across a synaptic cleft, when really there are at least three sources of "synaptic" delay:

1. **Synaptic delay**: The time between an electrotonus attributed to presynaptic soma depolarisation reaching a particular synapse, and the consequent realisation of this at the affected postsynaptic membrane due to the diffusion of neurotransmitter across its synaptic cleft.

2. **Axonal delay**: The time between a presynaptic soma depolarisation, and the consequent electrotonus reaching a particular synapse coupled to the postsynaptic neuron, along a particular path through the variable length axonal branching of the presynaptic neuron.

3. **Dendritic delay**: The time between a presynaptic depolarisation at a dendritically coupled synapse, and the consequent electrotonus reaching the soma of the presynaptic neuron along a particular path through the variable length dendritic branching of the presynaptic neuron.

Biological synapses whose physiology has been extensively studied show very little variation in true synaptic delay, and observation indicates that variance in synaptic delay is evident to an even lesser extent amongst physiologically similar synapses when compared to physiologically heterogeneous synapses. Since the neurons used here are assumed to be reasonably homogeneous, the assumed principle components of total delay are axonal, and dendritic delay. The label *axonal delay*, as in [19] is used to denote this delay. For want of a better notation, it is worth remembering that this really refers to the aggregate delay from all mechanisms that can contribute to it.

Thus in accord with [25, 1], multiple axonal delays are employed. An axonal delay d_{ij}^k is formally defined as the time in ms between the firing of a presynaptic neuron i, and the onset of the consequent PSP at postsynaptic neuron j due to the delay associated with the path from neuron i to neuron j, through synapse k. Each postsynaptic neuron is connected to each of its presynaptic neurons by m synapses. Each synapse $1 \dots m$ implements a delay $d^1 \dots d^m$. And specifically in this research, each delay is different; $d^n = \Delta_{min}^{ax} + \Delta_{interval}^{ax} \cdot (n-1)$. Modifying Equation-5 to use multiple axonal delays obtains Equation-6:

$$x_j(t) = \sum_{i \in \Gamma_j} \sum_{k=1}^{m} w_{ij}^k \cdot \epsilon_{ij}^k (t - t_i^a - d_{ij}^k) \tag{6}$$

The single connections ij in Equation-4 are replaced by several axonally delayed connections ij^k. Which allows the firing time of the postsynaptic neuron to be described in terms of the firing times of the set of neurons Γ_j, presynaptic to neuron j, and their axonally delayed d_{ij}^k, weighted w_{ij}^k afferent connections to the postsynaptic neuron j. The neuron j is said to fire at time t when the superposition of the incoming PSP as described by Equation-6 cause the membrane potential to exceed some threshold ϑ *from below*, as modeled by Equation-3, and illustrated in Figure-3(f).

Previous work using networks of spiking neurons [1], has assumed that each neuron in the network is either exclusively excitatory, or respectively exclusively inhibitory, consequently only generating EPSP, or respectively IPSP. This is reminiscent of the postulates of Dale [5] and later Eccles [8], the latter of whom stated that the same chemical neurotransmitter is released from all the synaptic terminals of a given neuron. Assuming that a single neurotransmitter is either exclusively excitatory or exclusively inhibitory, then it would logically follow that so is a neuron. However, according to [7], there is empirical evidence to suggest that single neurons can use multiple transmitters. It is not apparent whether this evidence indicates that a single neuron may heterogeneously release both excitatory and inhibitory neurotransmitters at different synapses. With regard to this, the idea that neurons are either exclusively excitatory, or exclusively inhibitory, was adhered to in the experiments described in this report.

2.2 Formal Description of a Spiking Neuron Network

The idea to treat a neuron network as an acyclic directed graph was obtained from [16].

Let Ξ be a feed forward network of spiking neurons (FFSN), then Ξ is a finite, acyclic, weighed, directed graph $\langle V, E \rangle$ where the vertices V correspond to neurons and a weighted edge $w_{uv}^k \in E$ denotes a connection between neuron u and v via synapse k. The layers of Ξ, Ξ_{layers}, partition V:

$$\Xi_{layers} = \left\{ \begin{array}{rcl} \Xi_n & = & o \\ \Xi_{(n-1)} & = & h_{|h|} \\ & \vdots & \\ \Xi_2 & = & h_1 \\ \Xi_1 & = & \iota \end{array} \right\} h \tag{7}$$

Notice the quantifying sets: h for all hidden layer neurons, o for output neurons, and ι for input neurons, these will be used throughout the following sections. Additionally each $x \in V$ has some attributes associated with it:

$$(\forall x \in V) \quad (\exists\, x^\rho \wedge \exists t_x^a) \tag{8}$$

Where x^{ρ} is the membrane potential of x, and t_x^a is the actual firing time of x. Furthermore, the following propositions hold:

$$\begin{array}{l} \forall l \in \{\Xi_1 \ldots |\Xi_{layers}|\} \\ \quad \forall u \in \Xi_l \\ \quad\quad \forall v \in \Xi_{(l+1)} \\ \quad\quad\quad \forall k \in \{1 \ldots m\} \\ \quad\quad\quad\quad \exists uv^k \in E \end{array} \tag{9}$$

$$|E| = \sum_{l=2}^{|\Xi_{layers}|} |\Xi_{(l-1)}| \cdot |\Xi_l| \tag{10}$$

The latter proposition is necessary to ensure that there can be no more connections in the network than those quantified into existence by the former. Notice this is just the normal notion of partitioning a network into layers in the feed-forward sense, with the notable exception of m axonally delayed connections between each neuron tuple. Each edge has some attributes associated with it:

$$(\forall uv^k \in E) \quad (\exists w_{uv}^k \in \Re^+ \wedge \exists uv_{type}^k \in \{-1, 1\} \wedge \exists d_{uv}^k \in [\Delta_{max}^{ax}, \Delta_{min}^{ax}] \tag{11}$$

This says three things; firstly that every edge, equivalently synapse, has a weight w_{uv}^k, equivalently synaptic efficacy, associated with it. The reason for the weight being a member of $\Re^+ = \{0 \cup \Re\}$ is because real neurons do not have negative efficacies, but allowing weights to be 0 encourages selective disabling by the learning procedure of connections which only contribute noise. Secondly, it says that every synapse has an associated type; either inhibitory (-1) or excitatory (1), and thirdly, it says that each synapse has an associated axonal delay d_{uv}^k.

Ξ employs the response function of Equation-5, and is simulated as in §3.3, ensuring that all elements of V have a stereotyped constant threshold ϑ. By virtue of the simulation allowing neurons to fire only once in a single simulation run, and assuming that sufficient time has passed since the last firing of any neuron that all membrane potentials and hence thresholds have returned to some constant value, this allows any dependence on the previous firing times of neurons that including a threshold function would obtain, to be ignored, i.e. the threshold function can be replaced with the stereotyped constant threshold ϑ for all neurons.

2.3 Learning With the Spikeprop Error Minimisation Paradigm

Spikeprop is a back-propagation-like learning rule for networks of spiking neurons. It is analogous to standard back-propagation learning rules in the classical domain. For the derivation details see the original paper by Bohte et al [25].

The procedure of application to an FFSN is to calculate deltas for the output and hidden neurons, and then use these deltas to update the synaptic efficacies. i.e. network weights. The equation for calculating the output deltas is different to that for the hidden deltas, because for hidden deltas, the firing times of successor neurons must be taken into account, whereas output neurons do not have successor neurons, so for output deltas the difference between desired and actual output firing times is considered instead. Equation-12 describes the function used to calculate the delta δ_j for an output neuron j.

$$delta^{out}(j) = \frac{(t_j^d - t_j^a)}{\sum_{i \in \Gamma_j} \sum_{k=1}^{m} w_{ij}^k \cdot \epsilon_{ij}^k (t_j^a - t_i^a - d_{ij}^k) \cdot (1/(t_j^a - t_i^a - d_{ij}^k) - 1/\tau)} \tag{12}$$

The numerator is the difference between the desired t_j^d and actual t_j^a firing times of output neuron j. The denominator iterates over all predecessors $i \in \Gamma_j$ of neuron j, and all synaptic connections $k = 1 \ldots m$ between those predecessors and neuron j and computes a sum. The sum is a product of three things; the first component is the synaptic weight on the connection k between predecessor neuron i and neuron

j (w_{ij}^k), the second is the membrane potential as governed by the spike response function ϵ_{ij}^k at the firing time of j given the predecessor firing time of i, with a delayed synaptic connection k. The third component is the reciprocal of the difference in firing times of neurons j and i, assuming synaptic delay k, minus the reciprocal of the decay constant τ which governs the rate at which neurotransmitter released from the presynaptic membrane reaches the postsynaptic membrane.

The spike response function is indexed by ϵ_{ij}^k, this implies that the particular form of the spike response function depends on the individual connection under consideration, however, in this research all connections are stereotyped by one of two polarity opposed, but otherwise identical, response functions, because a term of Equation-5, ij_{type}^k, is from $\{1, 0\}$, and hence all this index does is denote the sign of the response. Equation-13 describes the function used to calculate the hidden layer neuron deltas δ_i.

$$delta(i) = \frac{\sum_{j \in \Gamma^i} \sum_{k=1}^{m} w_{ij}^k \cdot \epsilon_{ij}^k (t_j^a - t_i^a - d_{ij}^k) \cdot (1/(t_j^a - t_i^a - d_{ij}^k) - 1/\tau)}{\sum_{h \in \Gamma_i} \sum_{k=1}^{m} w_{hi}^k \cdot \epsilon_{hi}^k (t_i^a - t_h^a - d_{hi}^k) \cdot (1/(t_i^a - t_h^a - d_{hi}^k) - 1/\tau)} \tag{13}$$

The numerator pulls in contributions from the layer succeeding that for which δ are being calculated, and hence passes the difference in firing time between each neuron $j \in \Gamma^i$ succeeding neuron i, and neuron i, to the function ϵ_{ij}^k, and it uses this difference in the last term of the product chain. The denominator respectively pulls in contributions from the layer preceding that for which δ are being calculated, and hence passes the difference in firing time between the current neuron i and each neuron $h \in \Gamma_i$ preceding neuron i to the function ϵ_{hi}^k, and it uses this difference in the last term of the product chain.

Once the deltas have been calculated, the network weights can be updated using them. Equation 14 shows the function used to adapt the hidden to output weights.

$$\Delta w_{ij}^k = -\eta \cdot \epsilon_{ij}^k (t_j^a - t_i^a - d_{ij}^k) \cdot \delta^j \tag{14}$$

Where δ^j is the output delta, ϵij^k is the indexed response function, and η is the learning rate. Equation 15 shows the analogous equation for adapting hidden to hidden weights.

$$\Delta w_{hi}^k = -\eta \cdot \epsilon_{hi}^k (t_i^a - t_h^a - d_{hi}^k) \cdot \delta^i \tag{15}$$

Notice how similar it is to Equation-14, the only difference is in the indexing. Both equations were shown for consistency with the δ calculating equations, but can be expressed as a general weight update function for the sake of convenience:

$$WeightUpdate(i, j, k) = -\eta \cdot \epsilon ij^k (t_j^a - t_i^a - d_{ij}^k) \cdot \delta^j \tag{16}$$

The final Spikeprop network adaption algorithm $FFSNAdapt$, is shown in §A, see Algorithm-2, its input requirements are specified in Table-5.

3 Computer Simulation

This section describes the simulation of networks of spiking neurons in functional terms and then proceeds to introduce and describe mechanisms by which various types of spiking network can be implemented on computer.

3.1 Theory

For a definition of an FFSN see §2.2. To recap the basic components, an FFSN Ξ consists of, input neurons ι, hidden layer neurons h, and output neurons o forming a partition of the set of all neurons of $V \in \Xi$, and a set of weighted feed-forward connections between successive subsets of V, termed network layers. The operation of Ξ can be described by a function that depends on the number of neurons in the input and output layers of Ξ:

$$\Xi : \Re^{|\iota|} \rightarrow \Re^{|o|} \tag{17}$$

In general this is a function of the form:

$$\Xi : \Re^{n} \rightarrow \Re^{m} \tag{18}$$

If the neurons of Ξ sporadically fired on occasion, which happens in biological neuron networks, or were otherwise generally noisy, then an output neuron could fire before the first input neuron had fired. However, since the model entails that the network is noise free, and neuron thresholds are stereotyped and invariable, no output neuron can fire until after the first input neuron has fired. Output neurons can fire before a subset of the inputs fire however, if the distance in time between the firing of two or more subsets of the input neurons is greater than the time it takes for the effects of a firing input neuron to become evident at the output layer, and the earliest firing subset is stimulating enough to solicit an output response. But it is still certainly the case that every output neuron will fire only after the earliest firing input has fired. Some minimum propagation delay $\Xi^{\Delta}_{min} = (|\Xi_{layers}| \cdot (\Delta^{ax}_{min} + t^{inc}))$ is also assumed, which is a product of the number of layers in the network $|\Xi_{layers}|$, and the minimum axonal delay Δ^{ax}_{min} plus at least one internal time step t^{inc} since at $t = 0$ the PSP response function returns zero. Given this, no output can fire until the earliest input has fired, and Ξ^{Δ}_{min} has elapsed, hence the function describing the operation of Ξ can be refined:

$$\Xi : \Xi^{in} \rightarrow \Xi^{out} where (\Xi^{in} \in \Re^{|\iota|} \wedge \Xi^{out} \in \Re^{|o|} \wedge (min(\Xi^{out}) \geq (min(\Xi^{in}) + \Xi^{\Delta}_{min})) \tag{19}$$

A single supervised training epoch for Ξ can be decomposed into two steps:

1. An input instance $x \in \Xi^{in}$ in the domain of Ξ is used in its feed-forward simulation.
2. The image of x, $y \in \Xi^{out}$, in the codomain of Ξ, is used to adapt Ξ given the firing times obtained from the previous step.

Each training instance in the training set for Ξ is a tuple, the first element is from the domain of Ξ in the function shown above, and the second element is from the codomain of Ξ. A training set can contain many such tuples. Ξ can be used to solve many kinds of problems, but they all have one thing in common; they can all be expressed in terms of the function that Ξ implements, for if they could not be, then Ξ would not be able to solve the problem. The original problem must therefore be expressible in terms of a non-empty set of input instances, and for each of these input instances, there must exist a corresponding output instance, otherwise there is no information for supervised learning. The problem P can therefore be described by a function of the form $P^{in} -> P^{out}$.

However, the domain and codomain of P may not be equal to those of Ξ, and thus the input and output instances of P may not be amenable to direct use in the supervised training of Ξ. Therefore there exists two functions which predicate supervised learning of P:

1. The first function $\mathcal{T}_{in}^{encode}$ temporally encodes input instances from the problem domain into input firing times that the network can use in its simulation.
2. The second function $\mathcal{T}_{out}^{encode}$ temporally encodes corresponding output instances, so that desired output neuron firing times can be obtained that can be used in the supervised learning adaptation step.

It is assumed that ultimately, if training is successful, Ξ will be used for its intended purpose; solving the original problem P, or at least attempting to solve it by generalising from the training set to unseen examples. Therefore there must also exist a function $\mathcal{T}_{out}^{decode}$ which temporally decodes the outputs of Ξ for a given input instance, back into the codomain of the original problem. Note that actually this function can be an intuitive interpretation of the outputs of Ξ, by a human, so it is not necessarily the case that an explicit function need be contrived.

In summary, in order to train Ξ in a supervised manner, and use it to solve instances of P, there must exist a temporal encoder T_{in}^{encode} that converts from the domain of P, P^{in}, to the domain of Ξ, Ξ^{out}, and there must exist a temporal encoder T_{out}^{encode}, and its inverse, a temporal decoder T_{out}^{decode}, that together can, in either direction, convert between instances of the codomain of P, and the codomain of Ξ. Formally, these three functions can be qualified as follows:

$$T_{in}^{encode} : P^{in} \rightarrow \Xi^{in} \tag{20}$$

$$T_{out}^{encode} : P^{out} \rightarrow \Xi^{out} \tag{21}$$

$$T_{out}^{decode} : \Xi^{out} \rightarrow P^{out} \tag{22}$$

Given some problem P, for which the three functions defined above exist, the two training steps for an epoch of supervised training become:

1. An input instance from the problem domain $x \in P^{in}$, is temporally encoded with $\mathcal{T}_{in}^{encode}$, and used as the input to the feed-forward simulation of Ξ: $\Xi(\mathcal{T}_{in}^{encode}(x))$
2. The image of x, $y \in P^{out}$, in the codomain of P, is temporally encoded with $\mathcal{T}_{out}^{encode}$, to obtain desired firing times which are used to adapt Ξ given the neuron firing times obtained from the previous step.

Once successfully trained, Ξ can generalise to the solving of instances of the original problem P. Given an instance $x \in P^{in}$, a temporal encoder $\mathcal{T}_{in}^{encode}$, and a temporal decoder $\mathcal{T}_{out}^{decode}$, then:

$$P(x) \equiv \mathcal{T}_{out}^{decode}(\Xi(\mathcal{T}_{in}^{encode}(x))) \tag{23}$$

3.2 Temporal Encoding and Decoding

As discussed in the previous section, temporal encoders $\mathcal{T}_{in}^{encode}$, $\mathcal{T}_{out}^{encode}$ and a temporal decoder $\mathcal{T}_{out}^{decode}$, are required to train a network to solve a real world problem, since networks of spiking neurons operate within exclusively temporal domains.

The notation used for a vector of firing times for a set of neurons y used in this paper, is shown below:

$$y^{\vec{fire}} = \{y_1 \dots y_n | y_i = t_{y_i}^a\} \tag{24}$$

$\mathcal{T}_{in}^{encode}$ can be implemented using overlapping Gaussian receptive fields as in [1]. In this research temporal encoding is implemented directly through alternating phase differences, for more information see the discussion of encoding and decoding, pertinent to the domains and codomains of Moore machines (§4.3).

3.3 Feed-forward Spiking Networks (FFSN)

Employing the spike response model introduced in §2.1 a single iteration of the procedure for training an FFSN Ξ on a set of input instances $\mathcal{I}$, and corresponding desired output instances $\mathcal{O}$, assuming the existence of appropriate $\mathcal{T}_{in}^{encode}$and $\mathcal{T}_{out}^{encode}$, can be described as follows:

The current input instance $\mathcal{I}_i \in \mathcal{I}$ is temporally encoded with $\mathcal{T}_{in}^{encode}$ to obtain the firing times $\iota^{\vec{fire}}$ of the input neurons for the current iteration of the training procedure. Initially the firing times $(V-\iota)^{\vec{fire}}$ of all other neurons are set to $-\infty$ to indicate that they have not yet fired. This is achieved programmatically by assigning them a negative firing time. From a start time t^{start}, time is increased in arbitrarily discrete steps t^{inc}, and the membrane potentials of all neurons but the input neurons $q \in (V-\iota)$ are recalculated using Equation-6 at each step, ignoring contributions from neurons which have not yet fired. If the calculated membrane potential x^{ρ} of neuron x exceeds the threshold ϑ, and the actual firing time t_x^a *has not yet been defined* (is $-\infty$), then the actual firing time t_x^a of neuron q is defined as t. This continues until either the simulation duration t^{len} has been exhausted, or all the output neurons $o \subset V$ have fired. It is acceptable to terminate before t^{len} has been exhausted if all o have fired, because all neurons whose firing directly affected the firing times of o, will have already fired, and thus all the information required for the adaption phase is present. After the simulation phase, the Spikeprop learning rule described in §2.3 uses the vector of output neuron firing times $o^{\vec{fire}}$ to adapt the network to accord more with the desired output firing times $o^{desired\vec{fire}}$, which are obtained by temporally encoding the current desired output instance $\mathcal{O}_i \in \mathcal{O}$ with $\mathcal{T}_{out}^{encode}$.

The following listed Tables and Algorithms all belong to Appendix-B. The FFSN simulation phase is formalised in Algorithm-1, its input requirements are specified in Table-4. The FFSN adaptation phase is formalised in Algorithm-2, its input requirements are specified in Table-5. Finally, the FFSN training procedure is formalised in Algorithm-3, its input requirements are specified in Table-6.

3.4 Recurrent Spiking Networks (RSN)

3.4.1 General Architecture of RSN

Let $x_{start(n)}^{\vec{fire}}$ denote the firing times of a set of neurons x at the start of simulation epoch n, and let $x_{end(n)}^{\vec{fire}}$ denote the firing times of a set of neurons x at the end of simulation epoch n. $x^{\vec{fire}}$ is the firing times of a set of neurons x at the time implied by usage context.

A feed-forward network of spiking neurons with recurrent connections is a special feed-forward network where a subset of the input neurons ι are also members of the set of state inputs ϖ, aka recurrent inputs, the reason for calling these inputs state inputs is because their firing times contain all the information pertaining to the current state that the system is in. The firing times of the state inputs are defined as follows:

$$\varpi_{start(0)}^{\vec{fire}} = \text{Predefined} \tag{25}$$

$$\varpi_{start(n)}^{\vec{fire}} = \alpha(\omega_{end(n-1)}^{\vec{fire}}) \tag{26}$$

Equation-26 shows how the firing times of the recurrent input neurons ϖ at the start of simulation n, are a function (α) of the firing times of some set of neurons ω, termed the recurrent-from neurons, at the end of simulation $(n-1)$. α can be any function of the form $\Re^{|\omega|} \rightarrow \Re^{|\varpi|}$.

Figure-4 shows a typical RSN architecture. It illustrates how the state input firing times ϖ at the start of simulation n are a function α of the recurrent-from firing times ω at the end of simulation $(n-1)$. In this case ω happens to be all neurons from one of the hidden layers of the network, however, more generally, ω can be any subset of V. ω are called recurrent-from neurons because this is where the *recurrent* inputs are obtained *from*. The next section discusses the instantiation of function α that is used in the experiments of this research.

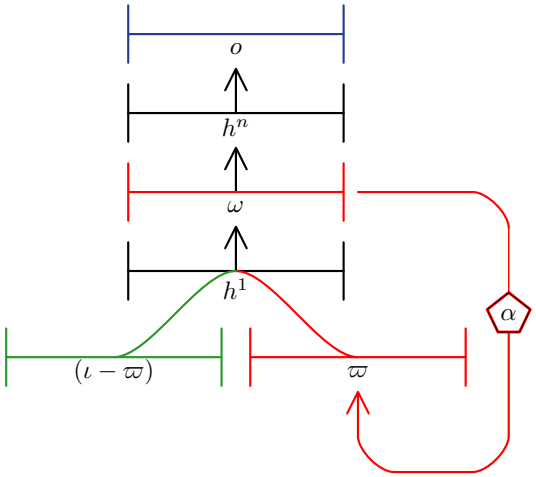


Figure 4: Typical RSN architecture.

3.4.2 RSN Training with Constant Delay and Monotonically Increasing Time

Here α implements a constant delay, and time monotonically increases with each sequentially executed simulation epoch. For the first simulation epoch, $t^{start} = 0$, and for each subsequent simulation epoch, t^{start} is incremented by the intra-input interval Υ. At the start of each simulation epoch, the inputs and desired outputs are encoded relative to t^{start}, i.e. t^{start} is added to them. Since the network update phase is only concerned with relative firing times, and not absolute firing times, it is independent of t^{start} and hence is applied as normal. Formally, α is defined as:

$$\alpha(x) = x + \Delta^{\alpha} \tag{27}$$

α realises a temporal translation of $\omega^{\vec{fire}}_{end(n)}$ by the delay constant Δ^{α}. In the experiments documented in this report the delay was always implemented using arithmetic addition. It is biologically implausible to claim that such an arithmetic addition can be utilised without first demonstrating that it can be achieved using networks of spiking neurons. Experimentation has shown that it is trivial to implement *exactly* arbitrary delays using an FFSN with one input and one output, so long as the desired delay does not exceed the temporal resolution at which the FFSN operates. Thus several of these trained networks can be used to delay the firing times of ω in parallel as is required by the experiments of §6 since none of these experiments uses a delay which exceeds the temporal resolution that an FFSN can be trained at. Since these networks *can* be trained if necessary, they do not have to be, and consequently the biological implausibility claim vanishes and arithmetic addition can be used instead for convenience and efficiency.

Figure-5 shows the first three steps of an operating RSN implementing the currently considered simulation mechanism. The initialisation parameters are displayed upper left; the simulation epoch count is set to zero in $n = 0$, the intra-input interval Υ is set to $15ms$, and the delay attributed to α is set to $10ms$. Above this is a table showing the inputs (I) for each simulation epoch, and the corresponding desired outputs (D). The desired outputs (D') relative to t^{start} for each output neuron, are shown in the boxes associated with them. Circles represent neurons.

The network considered has three layers: an input layer containing two state input neurons ϖ and one symbolic input neuron $(\iota - \varpi)$, a hidden layer containing two neurons which are also the recurrent-from neurons ω, and an output layer containing one neuron o. Three copies of the network are shown, corresponding to the first three simulation epochs given the input sequence (3,0,3). When $n = 0$ the state inputs are imposed by using predefined values. Notice also the monotonically increasing nature of t^{start} across the simulation epochs shown. As an example, in the first network, the input three, along with the imposed recurrent start times; both three, causes an output of twelve, whereas the desired output was fifteen.

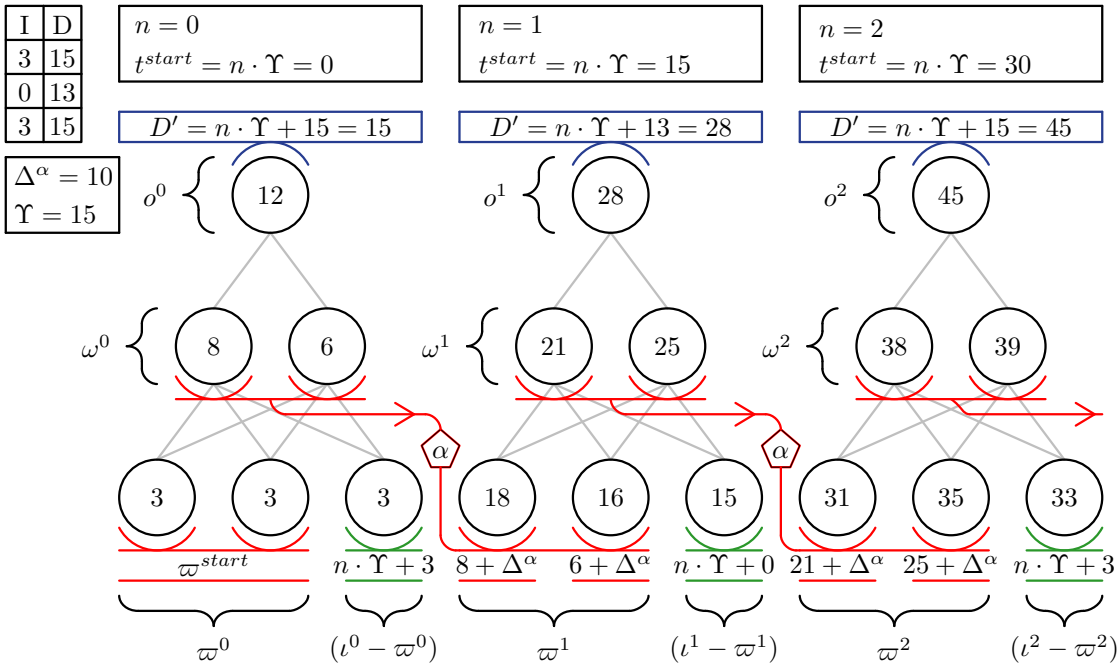


Figure 5: The first three steps in the operation of a recurrent network, where α implements a constant delay, and time monotonically increases with simulation epoch count. See text for further detail.

The most crucial observation is that the firing times of the hidden neurons ω^n at the end of simulation epoch n are translated in time by Δ^α by the function α to give the state inputs $\varpi^{(n+1)}$ at the start of simulation epoch $(n+1)$. And since the intra-input interval Υ is fifteen, $\varpi^{(n+1)}$ coincides nicely with the firing times of the symbolic inputs $(\iota^{(n+1)} - \varpi^{(n+1)})$ at the start of simulation epoch $(n+1)$.

This training procedure for RSN is formalised in Appendix-B, Algorithm-4, and its input requirements are specified in Appendix-B, Table-7.

3.5 RSN Training Via Temporal Expansion, i.e. Spikeprop-Through-Time-Networks (SPTTN)

This section describes how to simulate and train an RSN by expanding its operation through time. This procedure is coined Spikeprop-Through-Time and hence the collection of networks used to preform the expansion through time is referred to as a SPTTN (Spikeprop Through Time Network), but note that multiple networks are involved and contained within a single SPTTN.

The central component of an SPTTN $\Xi^{\Xi}_{\circlearrowleft}$ is a base RSN $\Xi_{\circlearrowleft}$. Given an input of length n, n copies of $\Xi_{\circlearrowleft}$ are made, stacked on top of each other, and sequentially simulated, incrementing t^{start} by Υ after each simulation. Adaptation deltas are calculated for each of the copies, and used to update $\Xi_{\circlearrowleft}$. The process of expanding $\Xi_{\circlearrowleft}$ through time via multiple copies, simulating an input sequence over the copies, then, conceptually contracting the copies, by updating $\Xi_{\circlearrowleft}$ using deltas calculated from each of the copies, and then deleting the copies, is repeated for the duration of the training regime.

Figure-6 shows the expansion through time of an example base recurrent network on an input of length 2. Labels are: recurrent-function α, symbolic inputs $(\iota - \varpi)$, state inputs ϖ, hidden layer neurons h, recurrent-from layer neurons ω, and output layer neurons o. Superscripts on labels denote copy membership. Subscripts on h index the hidden layers within a particular copy. Note that α is the same recurrent-function, implementing a constant delay, defined in Equation-27, that is used in the RSN simulation described in §3.4.2, which employed monotonically increasing time across multiple sequential simulation epochs.

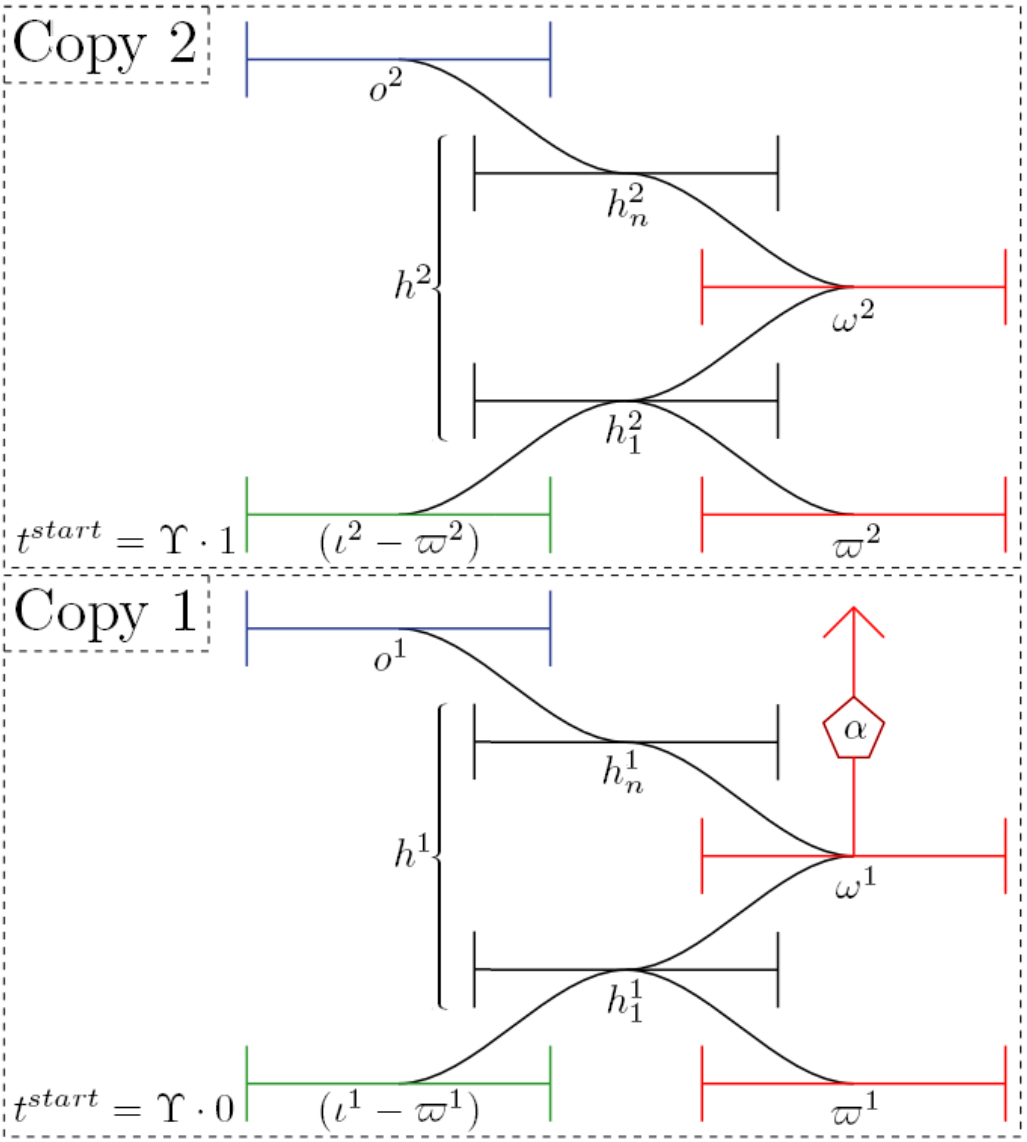


Figure 6: See Paragraph-3 of §3.5 for description.

Given a set of inputs $\mathcal{I}$, $|\mathcal{I}|$ copies of $\Xi_{\circlearrowleft}$ are stacked on top of each other as in Figure-6. The simulation for the first copy is different than for the succeeding copies because the first copy uses predefined recurrent start times, ϖ^1 in Figure-6, whereas the succeeding copies use the α delayed firing times of ω from the copy immediately preceding them.

For the first copy, the first symbol of the input string $\mathcal{I}_0$ is temporally encoded over the firing times of the symbolic input neurons $(\iota^1 - \varpi^1)$. The state inputs ϖ^1 are set to some predefined values, and the first copy is simulated with $t^{start} = \Upsilon \cdot 0 = 0$ so that all firing times of the first copy are relative to 0.

For copies $n \in \{2 \dots |\mathcal{I}|\}$, the recurrent function α is applied to the recurrent-from firing times $\omega^{(n-1)}$ of the succeeding copy to obtain the recurrent input times ϖ^n for the current copy. $\mathcal{I}_n$ is temporally encoded over $(\iota_n - \varpi_n)$ and made relative to $t^{start} = \Upsilon \cdot (n-1)$ by adding t^{start}, then the copy is simulated.

Given that an input string has been processed by the simulation of multiple copies, the Spikeprop deltas must be calculated, as in Equation-12 and Equation-13, for each of the copies for the adaptation phase. In a normal feed forward network, first the output layer neuron deltas are calculated, and then in reverse

order, the hidden layer neuron deltas are calculated. When the network is expanded through time, a similar procedure is applied; in reverse order from the outermost copy to the innermost copy, the deltas for the neurons in the output layer of the current copy are calculated, then in reverse order, the hidden layer neuron deltas of the current copy are calculated, calculating all deltas of one copy before moving on to the next. However, the procedure for calculating the deltas for the recurrent-from neurons ω in the outermost copy, is different from the procedure used in all the preceding copies. This is because all the preceding layers are connected to their succeeding layers by the recurrent function α, through ω, whereas the outermost copy has no succeeding layer.

For the outermost copy, the deltas for the output layer, then the hidden layers in reverse order, are calculated as if the copy was a standalone FFSN, but using desired output firing times made relative to the outermost copy's t^{start} by adding t^{start}.

For all copies preceding the outermost copy, the outputs, and then the hidden layers down to but not including ω are calculated as in a standalone FFSN, again using the desired output firing times made relative to the current copy's t^{start}.

In a normal FFSN, when calculating the deltas for a hidden layer, the firing times of the preceding and succeeding layer are used. However due to the connection from copy n, through ω^n and the recurrent function α, to $\varpi^{(n+1)}$ in copy $(n+1)$, it is as if there is an additional layer succeeding ω^n that must be taken into account. So actually, in addition to using the firing times of the layer succeeding ω^n in the current copy, the firing times of the first hidden layer in the succeeding copy $h_1^{(n+1)}$ are used too. The reason for using the firing times from the first hidden layer of the succeeding copy is that the derivative of a constant is zero, and α implements a constant delay, thus effectively entails the unification of ω^n and $\varpi^{(n+1)}$. The difference in firing times between $\varpi^{(n+1)}$ and $h_1^{(n+1)}$ contains the information that should be incorporated into the calculation of the deltas for ω^n. To achieve this, the delay constant Δ^{ax}, used by α, is subtracted from the firing times of $h_1^{(n+1)}$ and then, when calculating the deltas for ω^n, these temporally translated firing times are used as if they were simply another hidden layer succeeding ω^n. The rest of the hidden layer deltas are calculated as normal. Given that deltas have been calculated for the output and hidden neurons of all copies, the weights of the single network $\Xi_{\circlearrowleft}$ must then be adapted using them.

Given a normal FFSN Ξ, its weights are adapted by visiting its layers in reverse order, starting at layer $|\Xi_{layers}|$, and ending at the first layer. The weights between the current layer n and layer $(n+1)$ are adapted using the deltas from layer $(n+1)$. To update $\Xi_{\circlearrowleft}$ using all the information in the SPTTN $\Xi^{\Xi}_{\circlearrowleft}$, the copies of $\Xi^{\Xi}_{\circlearrowleft}$ are visited in reverse order from outermost to innermost. For each copy, in any order, the weight update values of the current copy are calculated in the normal fashion, but instead of applying them to update the current copy, they are used to update $\Xi_{\circlearrowleft}$ by adding each update value to the weight in $\Xi_{\circlearrowleft}$ that corresponds to the weight that would otherwise be updated in the current copy. The copies of $\Xi_{\circlearrowleft}$ are then scrapped and recreated for the next training pattern using the updated $\Xi_{\circlearrowleft}$.

Since multiple updates are applied in one batch, the learning rate η should probably be chosen smaller than for normal FFSN or RSN training, and intuitively a learning rate of (η/n) seems appropriate, where n is the number of copies in $\Xi^{\Xi}_{\circlearrowleft}$, and η is the learning rate that would be used to adapt $\Xi_{\circlearrowleft}$ if it were actually being adapted using normal FFSN or RSN adaptation. It is worth noting that for testing purposes, it is possible to perform updates on the copies of $\Xi_{\circlearrowleft}$ instead of $\Xi_{\circlearrowleft}$ itself, and then omit the last step described above where usually the copies of $\Xi_{\circlearrowleft}$ are deleted and refreshed from the recently updated $\Xi_{\circlearrowleft}$.

The following listed Tables and Algorithms all belong to Appendix-B. The SPTTN simulation phase is formalised in Algorithm-5, its input requirements are specified in Table-8. The SPTTN adaptation phase is formalised in Algorithm-6, its input requirements are specified in Table-9. Finally, the SPTTN training procedure is formalised in Algorithm-7, its input requirements are specified in Table-10.

3.6 Clustering Networks (CN)

The reason for needing clustering networks has to do with the extraction of induced structures from trained RSN, this is is explained in detail in §5.1. Essentially, a mechanism is needed to group together,

i.e assign a classification to, similar firing times observed in the state input layer of a trained RSN as it operates over a test set. This can be achieved using any unsupervised classifier that accepts real valued inputs, e.g. classical Vector Quantisation, however, clustering and classification via a CN was considered particularly applicable since the information for which classification was desired consisted of real valued neuron firing times, and since the state space that the neuron firing times belong to, as determined by the network, was thought to vary smoothly, CN clustering was considered more appropriate than other approaches because firstly CNs are good at classifying clusters and organising themselves if the target state space varies smoothly because of their utilisation of Gaussian-like kernels, and more importantly, since the core of a CN is an FFSN, it it is arguably more suited to processing temporal inputs than unsupervised classifiers that do not themselves operate in an inherently temporal manner, thus the clustering paradigm of [26, 1] is used here. Note that a CN is not something specifically designed to assist in the extraction of structures induced in the weights of RSN, it is an arbitrary unsupervised hierarchical classification tool applicable to many domains, however its main area of application has so far been the domain of computational neuroscience, for example CN clustering is used in [1] to address the so called binding problem in human recognition and memory.

The way that a clustering network is trained to classify its inputs into similar groups, is much like classical vector quantization (VQ). An FFSN internal to CN is fed the data to be clustered, temporally encoded over its input neurons, and simulated. The output neuron which fires first, the winner, indexes the the cluster of which the input is a member, this corresponds to the winning codebook vector in VQ. Like in VQ, the determinants of the winner must be adjusted so as to increase the probability of soliciting the same response given the same, and similar, input. The determinant of the winner in VQ is Euclidean distance to codebooks; nearest wins. The determinant of the winner in clustering with spiking networks is firing time; earliest wins. The update mechanism in VQ moves the winning codebook toward the current input, and to a lesser extent toward all other inputs, the magnitude being some inverse function of Euclidean distance, to encourage similar inputs to solicit a response from the same codebook, and consequently promote a topological ordering. The update mechanism in spiking net clustering, adjusts the network weights so that the winning neuron will fire even earlier given the same input, and as a result of the way FFSN work this encourages similar inputs to invoke the same winning neuron, and consequently promote a topological ordering.

The update rule of [26], which exploits temporal coincidence to achieve the aforementioned topological ordering, is shown below.

$$L(\Delta_t) = \eta((1-b)\cdot exp(-((\Delta_t - c)^2/\beta^2) + b)) \tag{28}$$

The value returned from this function is used to increase the synaptic weight between neurons that are presynaptic to the winning neuron, and the winning neuron; given that winning neuron w fires at time t_w^a, and one of its predecessors p fires at time t_p^a, employing an axonal delay of d_{pw}^k, then $\Delta_t = ((t_p^a + d_{pw}^k) - t_w^a$ is the difference between the onset of a PSP from p, and the arrival of that PSP at w. Feeding Δ_t into L returns a maximal response when the onset of an incoming PSP coincides with $(t_w^a - c)$. c is generally set up so that if a PSP onset occurred at time $(t_w^a - c)$, the course of the PSP relative to the firing of w implies that the two events are correlated and as a consequence the synaptic weight mediating the relationship should be increased, to in turn increase the potentiation of w so that it fires even earlier on subsequent simulations of the network. The learning function, constructed with some typical learning parameters is illustrated graphically in Figure-7.

A vertical translation of the function is affected by b so that the response for increasingly uncorrelated events converges to b. In the particular case of Figure-7, b is negative and hence parts of the function are below zero, enabling uncorrelated firing events to decrease the mediating weights, thereby enhancing learning by attenuating noise. The coefficient $(1-b)$ in Equation-28 scales the function, so that regardless of the translation that b affects, its maxima is always 1. c determines the presynaptic PSP onset time considered most important in determining the firing time of the postsynaptic neuron. β makes c more flexible so that the amount a synaptic weight is adjusted is an exponentially decaying function of its presynaptic PSP onset proximity to c. η in Equation-28 is just a standard adjustable learning rate.

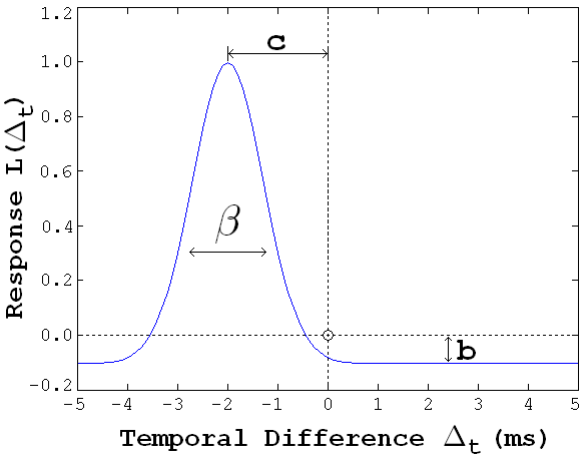


Figure 7: Spiking neuron network clustering learning function (Equation-28), with parameter realisations: $b = -0.1$, $c = -2$, $\beta = 1$, and $\eta = 1$

To avoid the under-representation of clusters during clustering, a surplus of output neurons is used. In [26] to avoid the over-representation of clusters, slow self-inhibition between output neurons is used to decrease the probability that a single output neuron will fire twice within a few iterations of the learning rule, however, it is not indicated how this might affect learning when similar inputs, proximally close in the input stream, are presented within a few iterations of the learning procedure over which the slow self inhibition is acting. If not dealt with appropriately this would presumably discourage learning.

As an alternative to this, it is suggested in [26] that the weights be initialised so that each output neuron needs at least one spike from every input neuron before it can fire. How this is achieved is not specified, and given the complexity of the network weight space, this was not attempted here. Hence the only guard against over-representation that was used in the experiments of this research, was appropriate parameter initialisation and the ability to repeatedly retry the clustering procedure, with the goal that the randomly initialised network weights, which correspond to initial codebooks, obtain the desired behaviour.

According to [26], if the weights are allowed to shrink and grow in an unbounded fashion, it can happen that some of the synaptic delay lines become inactivated. The proposed solution to this is to specify lower (w^{min}) and upper (w^{max}) weight bounds, and then clip any weight which exceeds a bound during updating to that bound. Following suggestions from the aforementioned, this was implemented with w^{max} set to the length of the first positive part of the learning function for a given set of parameters, a w^{min} of zero was used. They also suggest adding a saturation function to the learning equation, whose effect is to update weights that are near a bound less than those that are further from a bound, effectively realising a smaller learning rate for weights nearer the weight bounds. This was not implemented here.

The clustering network training algorithm is given in Appendix-B, Algorithm-8, its input requirements are shown in Appendix-B, Table-11. This procedure can be repeated multiple times for multiple random orderings of the training set as desired.

Once a clustering network has been trained, it can be used to classify subsequently presented input instances; the index of the winning neuron gives the classification. Therefore the number of outputs a clustering network has, determines how many unique clusters the network can classify input instances into, this is an upper bound, in practice, it will often be the case that some of the output neurons are not used, for example, if there are more output neurons than clusters. Choosing appropriate parameters, and some problems associated with clustering which are particularly relevant to the practice of extracting induced structures from trained RSN are discussed in more detail in §5.2.

4 Induction of Moore Machines using Networks of Spiking Neurons

In this section, theory put in a practical context, Moore machine induction using feed-forward networks of spiking neurons with recurrent connections is described in preparation for the next section, which puts theory into practice, detailing experimental results.

4.1 Definition of a Moore machine

Moore machines are a type of finite state machine, they are considered here and used in the induction experiments of this research. A Moore machine M is a finite directed graph, $\langle ME, MV \rangle$, its set of vertices MV are called states. Each state is labeled, and has an associated output symbol. There is a special state called the start state. Its set of edges ME, are called transitions. Each transition is labeled with a symbol, which can be presented to the Moore machine during simulation, hence is also known as an input symbol. The input symbols used by M can be replaced by an integer enumeration starting from 0, so can the output symbols, and so can the state labels. All M considered in this report are assumed to have undergone such a replacement procedure. Thus, there exists, for every M, a finite set of non-negative integers M^{in} containing all the input symbols it uses, and a finite set of non-negative integers M^{out} containing all output symbols it uses. M implements a function of the form $(MV, M^{in}) \rightarrow (ME, MV, M^{out})$; given a state in MV and an input in M^{in}, an edge in ME is transitioned on, producing a new state in MV, and an associated output in M^{out}. An example Moore machine is shown in Figure-8.

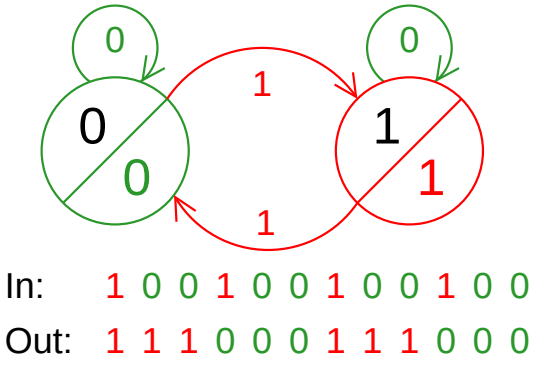


Figure 8: An example of a Moore machine and the output sequence (Out) it generates for an input sequence (In). Circles denote states. Lines denote transitions. The number in the upper-right of each state, labels it. The number in the lower-right of each state specifies its output value.

Given a sequence of n input symbols from M^{in} and a predefined start state, an operating M makes the start state the current state, and the first input the current input, then feeds them into the function $(MV, M^{in}) \rightarrow (ME, MV, M^{out})$, it outputs the third element of the result, makes the second element the new current state, accepts the next input symbol if there is one, and repeats this procedure until there are no more input symbols left, the first element ME is in the result to indicate which edge was transitioned on. Each step in the the operation of M is thus described by a function of the form $M^{in} \rightarrow M^{out}$. Enumeration over a sequence of inputs generates a sequence of pairs of the form $(x \in M^{in}, y \in M^{out})$, where y is the image of x in the function describing the operation of M. M can thus be characterised by the output sequence it generates for a given input sequence.

In less contrived terms, in order to parse a input string with M the current state is set to the start state, and the current input symbol is set to the first input symbol. The afferent edge from the current state labeled with the current input symbol is transversed, and the output symbol associated with the new state is output. The input is advanced to the next symbol, the new state made the current state, and the process repeated until the input string has been exhausted.

4.2 Beyond finite memory

The primary aim of this research is to show that trained RSN can go beyond finite memory. A trained RSN that has the capacity to go beyond finite memory is one which has properly induced the structure of a target MM in its weights, and remembers this structure indefinitely. For example, in Figure-8, a trained network that has gone beyond finite memory, emulating the MM shown, will be able to stay in either of the states of the MM, looping, given a sequence of 0 indefinitely, always getting the output

correct, but however long it stays in either of the states, upon seeing the input 1 it will successfully transition to the other state, which means that it never forgets the entire structure.

4.3 Encoding of Input and Output

Given a target Moore machine M, and a recurrent network $\Xi_{\circlearrowleft}$ with randomly initialised weights, a finite sequence of input-output symbol pairs is used to train $\Xi_{\circlearrowleft}$ to induce the structure of M in its weights, conceptually the inputs and outputs to the training procedure are respectively elements of M^{in}, and M^{out}, but actually they need to be converted to use with $\Xi_{\circlearrowleft}$hence this section describes how to translate from M^{in} and M^{out} to something more suitable for use with $\Xi_{\circlearrowleft}$, and how to translate from $o^{\vec{fire}}$ of $\Xi_{\circlearrowleft}$to M^{out}. The next section describes how to emulate M with $\Xi_{\circlearrowleft}$,

Referring back to §3.1, the domain and codomain allowed in the training sets of feed forward networks of spiking neurons Ξ were, and hence $\Xi_{\circlearrowleft}$ are, constrained by the domain and codomain of the functional description of Ξ shown below:

$$\Xi : \Xi^{in} -> \Xi^{out} where(\Xi^{in} \in \Re^{|\iota|} \wedge \Xi^{out} \in \Re^{|o|} \wedge (min(\Xi^{out}) \geq (min(\Xi^{in}) + \Xi^{\Delta})) \tag{29}$$

Given that each step in the operation of M can be described by the function $M^{in} \rightarrow M^{out}$, and the elements of M^{in} and M^{out} are arbitrarily sized tuples from $\mathbb{N} \cup 0$, if the domain of M were not modified, and elements from it were used in the training of $\Xi_{\circlearrowleft}$ directly, then during training, the latter constraint on the codomain of Ξ in the the above functional description would be violated if a training instance $(x \in M^{in}, y \in M^{out})$ were presented where $(min(y) < max(x))$. According to the theory of §3.1, to learn functions whose domains and codomains are not equal to those of the above functional description of Ξ, a temporal encoder $\mathcal{T}_{in}^{encode}$ is needed to encode the function inputs over the input neurons of the network, a temporal encoder $\mathcal{T}_{out}^{encode}$ is needed to encode the function outputs so that desired output firing times can be used in the network adaptation phase, and a temporal decoder $\mathcal{T}_{out}^{decode}$ is needed to decode network outputs, so that ultimately Ξ can implement the target function. In the context of M, the former two encoders are needed to train $\Xi_{\circlearrowleft}$, given only training instances of the form (M^{in}, M^{out}) consistent with the operation of M, and the latter decoder is needed to temporally decode the outputs of a trained $\Xi_{\circlearrowleft}$ to obtain elements of M^{out}, so that ultimately $\Xi_{\circlearrowleft}$ can be used to emulate M.

For the purposes of the experiments in this paper, $(\iota - \varpi)$ is partitioned into two disjoint sets; input neurons ι^s to represent the current input symbol, and reference neurons ι^r which always fire at the same time relative to t^{start} for any simulation run. The relative reference neuron firing times, and the number of reference neurons, are amenable to arbitrary specification.

$\mathcal{T}_{in}^{encode}$ temporally encodes the inputs of M, $x \in M^{in}$, over ι^s of $\Xi_{\circlearrowleft}$. It takes an integer input and converts it to a binary bitstring, then each binary bit of the bitstring is encoded as alternating high-low, or low-high firing times over a subset of the input neurons.

Input	Bit 1		Bit 0	
0	0	6	0	6
1	0	6	6	0
2	6	0	0	6
3	6	0	6	0

The table above shows how the integer inputs 0, 1, 2, and 3 are represented as a temporal binary encoding over two groups by $\mathcal{T}_{in}^{encode}$; where each group represents one bit, and within each group a bit is encoded as alternating *high* $(6ms)$ and **low** $(0ms)$ firing times, starting with *high* for logical 1, and starting with **low** for logical 0. Reference neuron firing times are not shown since they always fire at the same time relative to t^{start}. The scheme scales to any desirable number of neurons per bit. This allows the cardinality of ι^s to be made closer to the cardinality of ϖ, so that one type of input information is not overpowered by the presence of the other. The firing times used for encoding, *high* and **low**, are amenable to arbitrary specification.

It is desirable to use an even number of neurons per bit so that the average firing time over the neurons encoding each bit is the same for logical 1 and logical 0. The reason for this is that it ensures that at least initially, given random network weights, that the firing times of the succeeding neurons are going to be in approximately the same range as each other irrespective of the particular input, and the same will hold for their successors, and so on until the output neurons. This is important because there should not be a bias in the input firing times which make it harder to achieve certain outputs, which could happen if the more prominent firing magnitude in an odd number of input neurons, shifted the firing times of the neurons in any succeeding layers in a direction which resulted in a subset of the the network neurons being able to make use of only a small subset of the axonal delays for certain inputs.

After an input x from training instance (x, y) is encoded with $\mathcal{T}_{in}^{encode}$ over ι^s, and ϖ are set up, $\Xi_{\circlearrowleft}$ can be simulated. After the simulation, $\Xi_{\circlearrowleft}$ can be updated, but not without defined target firing times. These are obtained by temporally encoding the output y of training instance (x, y) with $\mathcal{T}_{out}^{encode}$.

The encoding mechanism used by $\mathcal{T}_{out}^{encode}$ is the same as that used by $\mathcal{T}_{in}^{encode}$, except that the values of *high* and **low** chosen for the bit encoding are scaled appropriately depending on the architecture of $\Xi_{\circlearrowleft}$. Typically only one output neuron per bit is used so that *high* and **low** correspond directly with logical 1 and logical 0 respectively. The reason for this is to make meaningful decoding of the outputs easier, since the encoding of target outputs determines how decoding of actual outputs must be performed after the network has been trained. With only one output neuron, $\mathcal{T}_{out}^{decode}$ can interpret each output neuron firing time as corresponding directly to a logical 1, or a logical 0, depending which of *high* and **low** the actual firing time is most close to. With more neurons it is not immediately obvious what $\mathcal{T}_{out}^{decode}$ should do in cases where some of the supposedly alternating firing times do not alternate, for example if the desired firing times for a group encoding a logical 1 with four neurons are $(26, 20, 26, 20)$, but the actual outputs are $(26, 26, 26, 26)$, or $(20, 20, 20, 20)$

As was the case for encoding inputs, temporal binary encoding of desired outputs should ideally be over an even number of output neurons. The reason is because there should not be a bias in the desired output neuron firing times which makes it harder to learn certain outputs, for example, it could happen that some of the encodings have an average firing time lower or higher than the rest, and because of a bias in the architecture of $\Xi_{\circlearrowleft}$it happens that the rest cannot make use of the full range of axonal delays as effectively, compared to those that are translated through time by some amount.

In the experiments of this paper, the number of neurons encoding each input bit were made equal to the cardinality of ω, and one neuron is used to encode each output bit, primarily because there was not a better decoding scheme available, and there were problems with learning when multiple outputs were employed (See §7.3 for elaboration).

4.4 Emulating a Moore Machine using a Recurrent Network

Assume a feed-forward network of spiking neurons with recurrent connections $\Xi_{\circlearrowleft}$, and a Moore machine M. To emulate M with $\Xi_{\circlearrowleft}$, $\Xi_{\circlearrowleft}$ must be trained to act like M. In M, the initial current state is set to the start state, in $\Xi_{\circlearrowleft}$emulating M, the initial state, as represented by ϖ, is set to some arbitrary predefined values ϖ^{start}. The current input symbol is temporally encoded across a subset of the the input neurons ($\iota^s \subseteq (\iota - \varpi)$) reserved for the this purpose (See §4.3), which is the same as presenting the first input symbol to M. Then $\Xi_{\circlearrowleft}$ is simulated, which is the same as transversing an edge in M. Then as a consequence of the simulation, the output for the new state will have been encoded over the outputs of $\Xi_{\circlearrowleft}$. Which is the same as outputting the value of the new state, after a transition in M. The new state itself, will be encoded across the the recurrent-from neurons ω by the end of simulation run n, and translated through time by the delay function α to give the new state encoded over ϖ at the start of simulation run $(n + 1)$, this is the new current state at time $t + \Upsilon$, which is the same as making the new state the current state, after a transition on some input, in M. This process is repeated until the input string has been exhausted.

The only information about the current state fed into the network externally is the start state ϖ^{start}, the subsequent state information is obtained from the network itself through recurrent connections. Given ϖ^{start}, and an input string, it is thus possible to emulate the parsing of a string by M with a suitably

trained $\Xi_{\circlearrowleft}$.

Each input symbol in the input string causes M, or the trained $\Xi_{\circlearrowleft}$, to output one output symbol. Therefore it is possible to compare the output of a trained network emulating M, with the output of the mechanical simulation of M itself over arbitrary input sets, consequently, the error rate of the trained network over some input string can be determined. If the error rate is zero, then the network must have induced some persistent structure in its weights that enable it to emulate M, at least for this particular input string. If the error rate is zero over multiple, randomly generated test strings of arbitrary length, which collectively exhaust the states of M, then it is increasingly probable that the trained network has induced some persistent structure capable of perfectly emulating M for any given input string.

4.5 Training a Network to Induce the Structure of a Moore Machine

4.5.1 Introduction

Given a feed-forward network of spiking neurons with recurrent connections $\Xi_{\circlearrowleft}$and randomly initialised weights, a target Moore machine M, a training set, and learning parameters, an induction training session can be executed, using one of the training paradigms described in §3.4, and §3.5. This section describes various considerations pertinent to this process.

Recall that $\Xi_{\circlearrowleft}$, set up to emulate the operation of M, will output exactly one output symbol when presented with one input symbol. And therefore it is possible to generate a string of input symbols, and corresponding desired output symbols, consistent with the behaviour of M. Several specifically crafted input-output symbol sequence pairs of this kind are used to construct the training set for M.

The constructed input-output symbol sequence pair (the training set) is not uniquely characteristic of M; for a given input sequence, there are an infinite number of Moore machines whose output behaviour is consistent with M. This is the the reason that clustering and extraction are performed later; to compare the structure of the extracted Moore machine with M, for equivalence. It is intuitively obvious that if the trained network gets no error on arbitrarily large test strings which fully exercise M, then whatever structure has been induced, if extracted, can be minimised to M, assuming the use only of inputs belonging to the transitions of M in the training and extraction phases. In practice however, guaranteeing that the network will always perfectly emulate the target MM is more of a challenge.

4.5.2 Theoretical Considerations

When a feed-forward network of spiking neurons with recurrent connections $\Xi_{\circlearrowleft}$ is presented an input symbol over ι^s, and current state over ϖ^n, it has to predict two things; the undelayed next state over ω^n, and its output over o. The firing times of $\varpi^{(n+1)}$ are obtained by applying the function α to the firing times of ω^n. In this research, α performs a linear deterministic operation, and applying α to the firing times of ω^n, just implements a temporal translation. Thus, the firing times of ω^n encapsulate the prediction of the next current-state for simulation run $(n+1)$.

Thus, $\Xi_{\circlearrowleft}$ can be thought of in terms of two feed forward networks. The first, given as input the current state and input symbol over its inputs, predicts the next state over its outputs. The second, given the outputs of the first as input, predicts the output value attributed to the next current state. For any Moore machine M, a unique temporal representation can be contrived for each state, for all possible inputs, and all possible outputs, assuming the availability of enough neurons to distribute the encoding over, so that the minimum separation between encodings of any two distinct values, is not beyond the network's inherent discriminatory capacity. Thus, the first half of the network can be trained separately from the second to correctly predict the next state over its outputs when given a state and input symbol over its neurons, and the second half of the network can be trained separately from the first to predict the output value attributed to each state over its neurons. Once trained, these networks can be combined to emulate M because by virtue of their combination, the outputs of the first will be the inputs of the second, and after each input symbol, the next state can be obtained by applying α to the firing times of the neurons which were formerly the outputs of the first.

In summary, given M, a feed forward network with recurrent connections, conceptually split in half into two concatenated networks, each capable of non-linear prediction, and given suitable temporal representations for states, inputs, and outputs of the M, it is trivial to separately train the two conceptual halves so that the emergent network can successfully emulate the operation of M.

The aim of this research is *not* to test the non-linear learning capability of feed forward networks of spiking neurons Ξ, by making them learn weights which entail their use as arbitrary Moore machine emulators, by using a carefully constructed training procedure employing contrived representations of their states, as in the above discourse. The aim of this research is to investigate the *induction* of network weights which allow the trained network to be used as an emulator for a specific target Moore machine M, training *only* on input-output sequences consistent with the behaviour of M, and not providing any *external* information about the current state, except the start state. Henceforth, the former training procedure, shall be referred to as direct training, and the latter training procedure shall be referred to as induction training, where in both cases, training refers to training some network using the Spikeprop learning rule of §2.3 so that the resultant weight configuration of the network entails using it to emulate M by the procedure described in §4.4.

Exercising direct training only serves to test the non-linear capacity of Ξ, and prove for a particular feed-forward network of spiking neurons x, that there is enough temporal discriminatory power bound in the weights and simulation parameters attributed to x to represent M.

It could happen that in practice, induction training requires in general, more network power than direct training to achieve a reasonable success rate. So that the relationship between successful direct training for a given feed-forward network architecture, and target Moore machine, and successful induction training, is non-linear. It is assumed however that some kind of positive correlation exists between the success of direct training, and the success of induction training, for a particular feed-forward-network architecture and target Moore machine.

For a particular target Moore machine M, there will in practice be some minimum-network-size prerequisite to successful training, given invariable auxiliary parameters, as a consequence of the finite temporal discriminatory and representational power which arises from discretising network operation and time within the simulating computer, and associated numerical precision limitations. Thus it might be of benefit, given M, to check the success rate of direct training, before embarking on induction training, using a particular feed-forward architecture, as if the success rate of the former is low, then the success rate of the latter is probably going to be even lower, this postulate is based on the empirically observed triviality of direct training compared to the relatively contrasting observed, and putatively onerous, induction training. In the induction training experiments of §6, the network architecture's power is not first tested by measuring the success of a direct training run using it, but it was worth mentioning.

Although the spiking neuron network model used in this paper is actually quite biologically unrealistic when compared to the complexity of real networks of neurons in biological organisms, it is desirable **not** to confound the situation further by making assumptions that would render the model *completely* biologically infeasible. Thus, when the operation of a recurrent network is continued through time as in §3.4.2, and §3.5, some retrospectively obvious constraints on the simulation interval Υ become apparent: Υ must be larger than the time it takes for the threshold of a just-fired neuron to pass through its absolutely refractory period and return to its baseline value, and Υ must be larger than the simulation duration $(t^{end} - t^{start})$.

The former requirement is necessarily consequent of using an invariant-threshold model, and the latter is necessarily consequent of the intra-input nature of network updates. In theory, $(t^{end} - t^{start})$ should be greater than or equal to the maximum output firing time that can occur for a given network architecture and parameters. Formally, in order to guarantee that all neurons that *can* possibly fire, *will* fire during the simulation, $(t^{end} - t^{start})$ should equate to $(max(\iota^{\vec{fire}} + ((|\Xi_{layers}| - 1) \cdot (\Delta^{ax}_{max} + \Delta^{\tau})))$; allowing time for the latest firing input, relative to t^{start}, to propagate to the outputs via the putatively unlikely path employing the synapse bearing Δ^{ax}_{max} between each layer. Δ^{τ} is some positive value which adds an additional delay since some neurotransmitter must diffuse over the synaptic cleft before the post synaptic potential at each neuron can exceed its threshold, and the dynamics of neurotransmitter are governed by by Equation-5. It is worth noting that this assumes that the action of a single EPSP can potentiate the firing of a neuron, whereas in biology this is unheard of. This is rationalised by claiming that a single

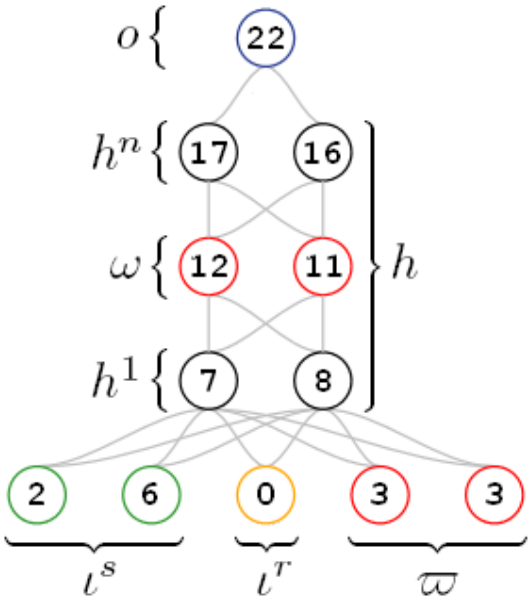


Figure 9: See text for details

synapse in the model can represent multiple, stereotyped, synchronously firing synapses in a biological system, which together have the capacity to potentiate a response [16], and assuming they are updated identically.

4.5.3 Architectural Considerations

Figure-9 shows a five layer recurrent network, the circles represent neurons, and the lines represent connections. It has two inputs ι^s to represent the current input symbol, a single reference input ι^r which always fires at t^{start}, and some recurrent inputs ϖ which represent the current state. The recurrent connections from ω to ϖ are not explicitly shown. There are two hidden units in the first hidden layer h^0, two in the second hidden layer ω; the recurrent-from layer, two in the third layer h^n; where $n = 3$, and a single output neuron in o.

The average firing times of the network neurons over a series of typical input sequences, are implicative of temporal bounds within which the firing of neurons within a particular layer are most probable. Assuming that the times shown in Figure-9 are typical, approximately $5ms$ elapses between the firing of neurons in each subsequent layer, and hence the output firing times naturally cluster around $\approx 20ms$, hence, for this network it is probably a good idea to set the target output firing times, encoded as in §4.3, relative to $\approx 20ms$. Empirically-based selection of appropriate output firing times, can be performed by observing the average firing times of several untrained networks with initially random weights. This assumes that the network weights are initialised in such a way that they are within the sort of range that the network is naturally inclined to use, and of course, it is not so easy to determine what kind of range this is because any empirical study is likely itself to be biased by the initial weight setting. Minimising the error over a test set, by selecting parameters and repeatedly retraining on some marginally hard but learnable classification problem until a low error rate is obtained, i.e intelligent trial and error, would probably be a good approach to optimise weight initialisation ranges.

It turns out the latter constraint on Υ is somewhat illusory speaking strictly in terms of correctness of the simulation. It is easy to see that if ω always fire at approximately $((average(\omega^{\vec{fire}}) - average(\varpi^{\vec{fire}}))$ relative to t^{start}, then by appropriately choosing Δ^α the firing times of ω at simulation run n, perturbed by α, can be made to occur proximal to t^{start} through ϖ at simulation run $n + 1$. Assuming a desire for $\varpi^{\vec{fire}}$, temporally proximal to t^{start} , for any desired Υ, Δ^α can be set to $\Upsilon - (average(\omega^{\vec{fire}}) -$

$average(\varpi^{\vec{fire}})$), trivially delaying the firing times of ω by the difference between Υ and the average separation of ϖ and ω.

Observing Figure-9, it is apparent that approximately $10ms$ elapses between the firing of ϖ and the consequent firing of ω. Given that the firing times of ω are likely to cluster around $t^{start} + 10ms$, if Υ is set to $10ms$, then this time is equivalent to t^{start} at simulation run $n + 1$. Consequently, and by virtue of their construction, the firing times of $(\iota - \varpi)$, in this case, will occur within approximately the same temporal window at simulation run $n + 1$, relative to t^{start}, as the firing times of ϖ.

Although the above paragraph discusses violations of the aforementioned Υ constraints, it serves to illustrate that the delay implemented by α merely serves a conceptual purpose; that of indicating to an observer that there has been a sufficiently long temporal interval between the presentation of one set of inputs to allow an update before the presentation of the next. Since the computer is not bound by absolute time it matters only that an update is performed between two simulations, and that the relevant firing times of all determinants of the outputs, and t^{start}, are setup correctly at the start of each simulation run.

In a Markovian manner, $\varpi^{\vec{fire}}$, and $(\iota^s)^{\vec{fire}}$ are determinants of the next state and its output, and it is assumed that the capacity of the network to properly make use of these determinants, such that they are both involved in determining the firing times of subsequent neurons, is a function of their relative temporal proximity. The reason is because if they were not temporally proximal, then there is a greater chance that only one of the above will be solely responsible for determining the firing times of successor neurons, because if one has larger weights and fires first, it can cause successors to fire before the others fire. And hence correlated average firing time onset over ι is a desirable property of appropriate parameter selection.

If Δ^t and Υ are inappropriately chosen, the average firing times of ϖ will occur out of phase with the average firing times of ι^s. Furthermore, if because of inappropriate assignments to the aforementioned variables, ϖ is always temporally shifted by some constant relative to t^{start}, this will presumably cause the consequent firing times of ω to be shifted similarly, assuming some reasonably strong, relatively invariable, temporal coupling between ϖ and ω. Thus after repeated iterations, $\varpi^{\vec{fire}}$ might gradually shift out of phase relative to the firing times of ι^s, decreasing the temporal coupling between ϖ and ι^s. Unless the network is able to self-adjust to balance this. It is not unreasonable then to assume that if the network has to induce some kind of scaling of $\varpi^{\vec{fire}}$ to balance this, then the learning capacity of the network whilst this is occurring might be attenuated to some extent. And depending on how much the path through weight space taken during learning is dependent on steps taken early in the path, inappropriately chosen network parameters could even negatively effect the whole learning procedure. Whereas if the parameters are chosen appropriately so that the firing times of ϖ and $(\iota - \varpi)$ are sufficiently temporally proximal, and no gradual temporal shift occurs, it is perhaps probable that the learning rate will be faster since the network will not have to deal with the additional problem of adjusting its weights to offset the gradual temporal shift of the firing times of ϖ relative to those of t^{start}.

Concluding, it is thus desirable to make intelligent estimations for network parameters based on the observed operations of typical instantiations of such networks, knowledge of their operation, and recognised theoretical problems. However, although this has seemingly obvious benefits over choosing network parameters randomly, the extent to which this knowledge can be applied with obvious benefit is unknown without first understanding it in the context of a detailed understanding of how recurrent networks' actually induce structures, their capacity to deal with noise, temporally shifted inputs, and so on. Consequently, future development should probably investigate the considerations outlined in this section in more detail.

4.5.4 Selecting Appropriate Training Sequences

In induction training, since a contrived current state is not presented with each input, the states must be learned as a side effect of training on a training set in a normal feed-forward manner, using appropriate recurrent connections to define some of the inputs at each step. Thus, the most important factors to success are: the training set, and the network architecture.

Since this kind of research has never been done before with spiking neuron networks, it is not apparent what kinds of input sequences will bias the induction process positively. Research using classical artificial neural networks have used several short training sequences which exercise characteristic cycles and features of the target MM, consequently, as a starting point, so has this research.

As an example, consider the Moore machine of Figure-8. Since it oscillates between states 0 and 1, on input 1, a short sequence will be included in the training set that consists of the symbol 1repeated. There are also self-referential connections looping on each state, hence some of the training sequences of the training set should exercise these loops. Since it is desirable to use several short sequences in a training set, the sequences must somehow be separated. This is achieved by resetting the system between presenting training sequences, this can be achieved in at least two ways.

The first reset mechanism is termed an explicit reset since $\varpi^{\vec{fire}}$ is explicitly set back to ϖ^{start} between the presentation of each subsequent training sequence, and t^{start} may or may not be reset to zero. The second reset mechanism is termed an implicit reset since no firing times are imposed on ϖ, instead, a special implicit reset symbol is declared, and defined in the target Moore machine by adding transitions from every state on the reset symbol to the start state. The aim is get the network to recognise implicitly that consistently setting $\varpi^{\vec{fire}}$ to some stereotyped values upon seeing the reset symbol, will improve the error over the entire training set. Note that the start state itself is self-referential on the reset symbol, and in the experiments of this paper, it was always chosen to be larger than any other input symbol used by the target Moore machine.

The second approach is more desirable than the first. It is believed that this has something to do with the way that the induction works, the idea is that by repeatedly imposing ϖ^{start}, there is increased probability that it will be adopted by the network as one of its states, and if it is adopted and it is not typical of the states that the network would naturally be more inclined to use given its setup parameters, then it could attenuate the learning capacity of the network, and on the other hand, if it is not adopted, then it is probably contributing a significant amount of noise and pulling the network away from the region in weight space which it is has a higher affinity for given its setup parameters. Hence the preference for an implicit reset symbol. ϖ^{start} still has to be imposed for the first symbol of the first sequence in the training set, but the rest of the training set is a concatenation of the several sub-sequences, each separated by one or more implicit reset symbols.

Furthermore, usually a "reset run" is presented at the start of a long training sequence when the second approach is used so that there is ample opportunity to use this run to move away from the imposed start state to a place in weight space that the network has a higher affinity for, well at least, this is the theory, and in practice it appears to work.

4.5.5 Network Parameters and Weight Initialisation

In classical networks of spiking neurons, the weights are conventionally initialised to random elements of $\Re$ from the interval $[0, 1]$. For the clustering networks of §3.6 this is still applicable, but for general feed forward networks of spiking neurons, it appears that this is not applicable when using the Spikeprop learning rule; learning of simple non-linear training sets appears to fail if the threshold is not sufficiently high, and the weights are not sufficiently scaled. The problem was first identified in [23], and it was from this thesis that the weight setting, neuron threshold parameters, and the learning rate, for the majority of the experiments of this paper were obtained. The threshold used in these experiments is almost always 50, and the are weights almost always initialised to random elements of $\Re$ from the interval $0, 10$. Whatever weight setting is chosen, the weights must be initialised so that the neurons in subsequent layers are sufficiently excited by those in their previous layer that they fire, otherwise the network would be unusable. There is no equivalent in classical neural networks to the non-firing of a neuron in this sense.

A considerable amount of unsystematic auxiliary analysis was done to try and determine why such weight settings were required. It seems as though the learning mechanism is not invariant to a simple proportional attenuation of the weights and thresholds. A much more rigorous and formal analysis should be carried out to obtain a sufficient explanation of this, as the weight setting appears to be fundamental

to the success of the learning phase.

Each neuron is connected to all successor neurons by a number of synaptic delays, in the experiments of this paper, 16 synaptic connections per connected neuron pair were used, the axonal delays were explicitly set over the 16 synapses between every connected neuron pair to $1ms \ldots 16ms$.

It is worth pointing out that empirical evidence suggests that the learning rate has to be unusually high to get fast convergence with spiking neurons when compared to classical artificial neuron networks. It has been suggested that an learning rate of 1 be used, and in practice this often works and converges fast, whereas it appears that in classical networks, this would be considered too high, and may even be a determinant to learning.

The experiments reported here all use a dynamic learning rate, this attempts to detect oscillatory behaviour and the finding of a plateau within the error space. The user specifies an initial learning rate, the number of epochs to average over, an oscillation threshold, a plateau threshold, an oscillation-counter-coefficient, and a plateau-counter-coefficient. The absolute error is stored for each epoch until the specified number of epochs have accumulated, then an oscillation is said to occur if, over those epochs, the error alternates from positive to negative or the average difference is greater than the oscillation threshold, a plateau is said to occur if the average error difference over the specified number of epochs is less than the plateau threshold. The action to take upon detecting oscillation or plateau, is respectively to decrease the learning rate by multiplying by the oscillation-counter-coefficient, or increase the learning rate by multiplying by the plateau-counter-coefficient.

5 Extraction of Moore Machines from a Trained Recurrent Network

The firing times of the state input neurons of an RSN are thought to correspond to states of a structure represented by that RSN, obtained by some training regime. This section explains how the firing times can be examined and used to extract the structure than the RSN is inherently using.

5.1 Extraction Procedure

Using the ideas outlined in the last few sections, a recurrent network $\Xi_{\circlearrowleft}$, can be trained on a training set consistent with the operation of a target Moore machine M. If training is successful, and subsequent testing on a test set, randomly generated from M, is also successful, then it is probable that the weights have induced some persistent structure capable of emulating M to some extent. It is interesting to attempt to determine the exact nature of this structure, and hence this section describes how to perform what is called clustering and extraction.

In the clustering phase, $\Xi_{\circlearrowleft}$ is made to emulate the action of M over a randomly generated test string derived from M, and all $\varpi^{\vec{fire}}$ used in the emulation are recorded. As explained above in §4.4, $\varpi^{\vec{fire}}$ encodes the current state during the presentation of sequential symbols of the input sequence during the training phase, so by storing all $\varpi^{\vec{fire}}$, this procedure stores all of the states used by the network over some test set. The idea is that groups of the states actually correspond to single states in some automata space, that is, there are *clusters* of $\varpi^{\vec{fire}}$ which can be grouped together and given unique identification, and that hopefully these uniquely identified groups correspond to the unique states of an induced structure embedded in $\Xi_{\circlearrowleft}$. Ultimately it is hoped that this induced structure is similar or identical to M. So it is desirable to be able to extract the induced structure and compare it to M. Thus, the list of $\varpi^{\vec{fire}}$ are fed into an unsupervised learning mechanism that is capable of grouping $\varpi^{\vec{fire}}$ that are sufficiently similar to each other, this is alternatively known as clustering.

There are many clustering mechanisms that could be employed for this task, in the experiments of this report, the spiking network clustering mechanism of §3.6, and standard K-means clustering are used, but since the former is supposedly a novel paradigm, particularly amenable to processing temporal inputs, it is of primary interest, and thus is the primary paradigm discussed here. Note the distinction between states in M, states in the induced automata, and clusters in ϖ space. If the clustering procedure is perfect, then there will be no under-representation and no over-representation of states in the induced automata, by the clusters generated by the clustering procedure, and therefore there will be a one-to-one relationship between the clusters in ϖ space generated by the clustering procedure, and the states of the induced automata.

Assume a clustering network CN, trained as in Algorithm-8 over the list of $\varpi^{\vec{fire}}$ generated from a recurrent network $\Xi_{\circlearrowleft}$. Given any $\varpi^{\vec{fire}}$ obtained from the operation of $\Xi_{\circlearrowleft}$, CN will produce a classification index, which is the index of the winning neuron. An attempt can then be made to extract the structure induced in the weights of $\Xi_{\circlearrowleft}$. Its success will depend on how well the clusters of $\varpi^{\vec{fire}}$, determined by CN, relate to the structure in the weights of $\Xi_{\circlearrowleft}$, which dictates its behaviour. Given a randomly generated test string, a contrived test string, or an interactively input symbol sequence, a series of inputs can be presented to $\Xi_{\circlearrowleft}$, such that its outputs, the clustering of its $\varpi^{\vec{fire}}$ by CN, and the input presented at each step, can be used to enumerate, respectively outputs of states, states, and transitions between states. Repeated application of a simple procedure is executed to enumerate the states of an induced automata, this is shown below:

1. At the start of the input sequence $\varpi^{\vec{fire}} \leftarrow \varpi^{start}$.
2. Encode the current input s with $\mathcal{T}_{in}^{encode}$, over ι^s of $\Xi_{\circlearrowleft}$.
3. Process $\varpi^{\vec{fire}}$ with CN to obtain the index x of the winning neuron.
4. Simulate $\Xi_{\circlearrowleft}$in a feed forward manner.

5. Decode the output of $\Xi_{\circlearrowleft}$ with $\mathcal{T}_{out}^{decode}$ and store this value in y^{out}.
6. Process $\omega^{\vec{fire}}$ with α to obtain the new state across $\varpi^{\vec{fire}}$
7. Cluster $\varpi^{\vec{fire}}$ with CN to obtain the index y of the winning neuron.
8. Record that there is a transition from state x to state y on input symbol s.
9. Record that the output of state y is y^{out}.
10. If termination criteria has been met, exit the loop and finish.
11. Set s to the next input symbol in the input stream.
12. Goto step 2.

There is no real need to re-obtain the winning neuron index of the new $\varpi^{\vec{fire}}$ with CN at step three, after the first time round the loop, as the index of x will be the same as the index of y from the previous time round the loop. Note that at step six, $\varpi^{\vec{fire}}$ is set to the new state, so that at step two after looping, $\varpi^{\vec{fire}}$ represents the current state.

If a transition has not been defined for a particular state x on a particular input s, then given that the current input is s, and the current state is x, in step seven, the procedure will either return a y which corresponds to an old state, or a y which corresponds to a new state. Thus the procedure will continue to find new states and connections between states as it iterates through the input sequence of the test set. Then it is easy to contrive some termination criteria for step ten, which will halt the procedure when transitions have been defined for every possible input on each known state, so it is known that no new states will be discovered and what has been discovered is complete.

The procedure above is not so much of a concrete algorithm that must be adhered to, but more of an informal description of how to go about extracting an induced structure from a trained recurrent network given some clustering mechanism. The clustering mechanism used in steps three and seven can be replaced by any clustering mechanism which, produces a classification for each input instance, gives the same classification for similar instances, and gives different classifications for dissimilar instances, where similarity is defined by some metric. A couple of examples are traditional self organising maps, and K-means clustering, where in the latter "winning neuron" is replaced with "closest codebook".

Considering the extraction procedure, an additional benefit to using the implicit reset paradigm discussed in the last section becomes apparent. If a reset run is present in the input sequence used at the start of the extraction procedure, and assuming the network has at least induced the self referential connection on the reset symbol at the start state of the induced structure, which corresponds to the start state of M, then the reset run can be used to find the start state by presenting the reset symbol until the index returned from clustering is always the same for some predefined number of presentations, that is, the self-referential connection has been found on the reset symbol, implying the start state. The start state is a good place to start the extraction procedure from. Of course, if this were automated, and the start state was not found within a certain amount of presentations of the reset symbol, then the procedure should give up and start enumerating the states in the ordinary manner.

The most frequently observed problems associated with extraction, are artifacts of the prerequisite clustering phase.

5.2 Problems with Clustering and Extraction

The temporal clustering mechanism discussed in §3.6 is subject to problems that are analogous to common problems in most unsupervised learning paradigms. Choosing appropriate numbers of outputs, and selecting appropriate learning parameters, depends on the learning task, for example, if there are lots of small clusters in the target domain, then a relatively narrow learning function width should be used, and lots of output neurons should be employed, and if there are a few big clusters then fewer output neurons are probably necessary, and a wider learning function width probably advisable. There can be cases

where a big cluster is composed of lots of small clusters, so the exact learning parameters depend on the desired observation resolution and the exact problem, thus a considerable amount of trial and error, and intelligent reasoning, should be used to attempt to discern reliable parameters. Sometimes however, the problem can be beyond the resolution of the clustering mechanism used here. For example, if clusters are heterogeneously sized, then obviously some compromise will have to be come to with respect to the width of the learning function. But more insidiously, the target domain might be populated by what are called complex clusters whose detection entails the use of more sophisticated clustering mechanisms as in [**?**]. For a more detailed discussion of this see §5.2. Assuming that appropriate parameter selection is not going to be achieved with ease all of the time, and that the other problems might arise, this section discusses how the ramifications of these problems, and where appropriate, discusses how to recover from them. From the perspective of the Moore machine extractor, three notable problems can occur.

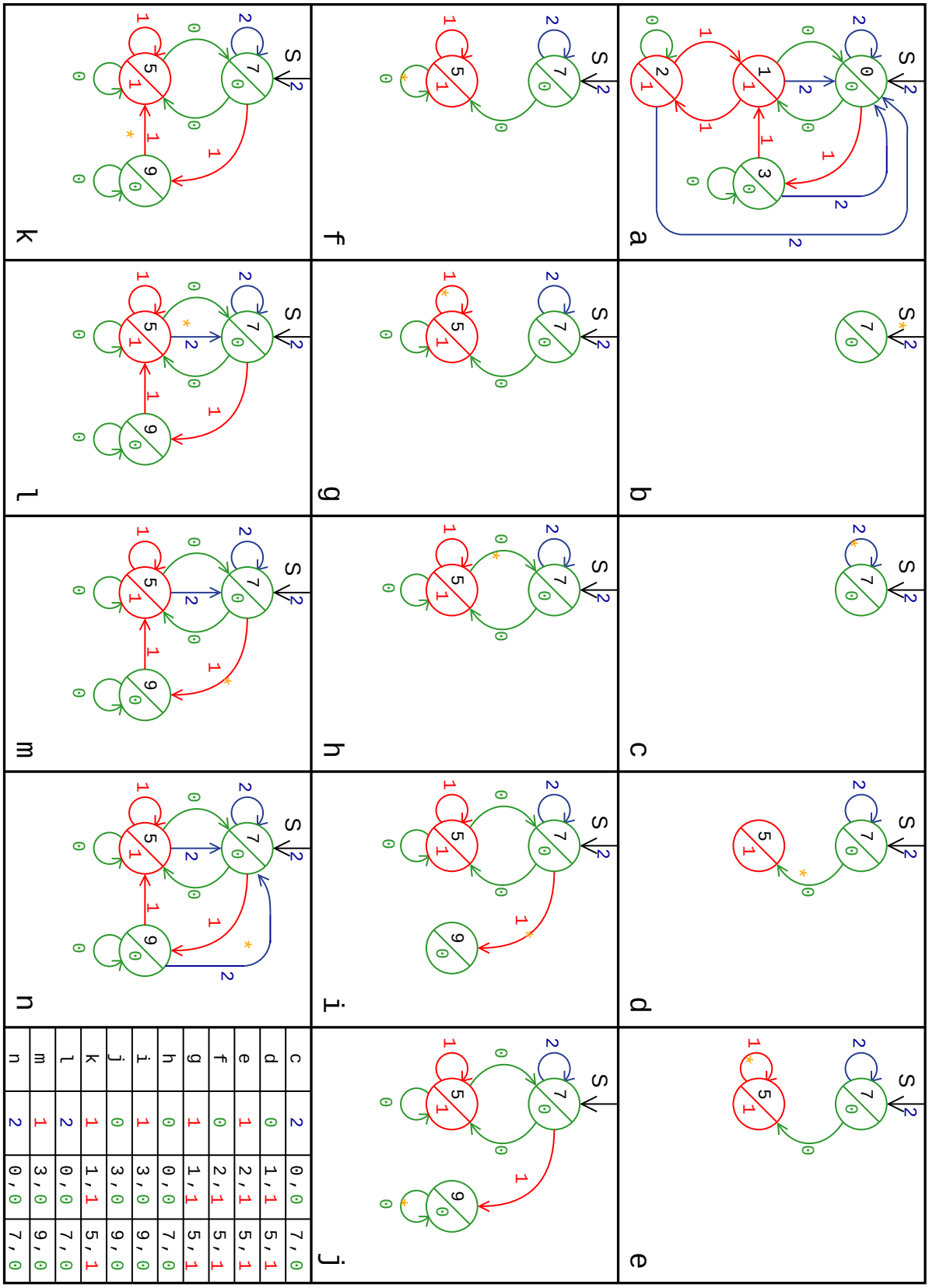
1. Firstly, it can happen that the clustering network training phase produces a network that over-represents; several similar $\varpi^{\vec{fire}}$ collectively having small variance, from the same cluster, are given different classifications by the clustering network, so that it appears as if as if there are several different clusters; different indices are returned from the clustering procedure for different subsets of the $\varpi^{\vec{fire}}$ from one cluster
2. Secondly, it can happen that the clustering network training phase produces a network that under-represents; several dissimilar ϖ having collectively relatively large variance, belonging to several different clusters, are given the same classification by the clustering network, so that it appears as though they all belong to the same cluster; the same index is returned for $\varpi^{\vec{fire}}$ belonging to several different clusters.
3. Thirdly, and most heinous, it can happen that the clustering network training phase produces a network with both of the aforementioned undesirable traits.

If none of the problems mentioned above happen, then the induced automata is amenable to trivial automatic extraction given a random input string that exhausts its state space.

If only the first problem happens, then the automata extraction can be automated. The extracted automata will be non-minimal, but otherwise correct, it can thus be piped through a Moore machine minimisation algorithm to obtain the actual automata.

If the second or third problem occurs, things immediately start getting much more complicated, because a particular winning output neuron can fire in the clustering network for $\varpi^{\vec{fire}}$ from different states in the induced structure, or different output neurons can be the winner for different $\varpi^{\vec{fire}}$ from a single state in the induced structure. So in the worst case, the clustering mechanism will refer to a single state in the induced structure by several different names, and it will also refer to several different states by the same name. If the extraction is performed automatically by the naive procedure described in [**?**] and these problems are present, then the resultant extracted Moore machine will most likely be nonsense; it can have non-deterministic transitions on input symbols, as well as having states with non-deterministic outputs.

Figure-10-**a** shows an example target Moore machine M, transitions are labeled with input symbol, and circles represent states; the number in the upper left of each state uniquely labels it, the number in the lower left of each state indicates the output attributed to that state. Note that this automata was not actually induced in reality. Insets b–n show a fictitious automata extraction procedure for a fictitious trained network that has successfully induced M. Again, circles represent states. The number in the upper left of each state in figures b–n is the index of the winning output neuron returned at that extraction step by the clustering network, and thus ideally correspond to unique state identifiers, the lower right numbers in each state is the output value associated with that state. At each step, the currently presented input symbol is indicated by labeling it with an asterisk *, and if a new states is discovered as a consequence of some transition then it will be shown for the first time in that step. The diagram should be read with repeated cross-reference to inset a so as to get a clear idea of what is happening.



c	2	0	0, 0
d	0	1	1, 1
e	1	2	1, 1
f	0	2	1, 1
g	1	1	1, 1
h	0	0	0, 0
i	1	3	0, 0
j	0	3	0, 0
k	1	1	1, 1
l	2	0	0, 0
m	1	3	0, 0
n	2	0	0, 0

Figure 10: **a** Example automata, **b-n** Example extraction steps. Bottom right - Extraction detail. See text for details.

The extraction proceeds in the manner described in §5. **b**: the input symbol 2 causes the state $(7, 0)$ to be discovered and become the new current state. **c**: since the start state is considered to be self-referential on the reset symbol 2 assume at this point that several 2 were presented and the target state was always $(7, 0)$ so the computer assumed it had found the start state as discussed in the previous section. **d**: input 0 entails the discovery of state $(5, 1)$. **e**: the input symbol is 1, but the clustering has under represented the states of the induced automata, and since it was assumed the trained network has successfully induced M, the clustering has also under-represented the states in M, so that states $(1, 1)$, and $(2, 1)$ in M are now both represented by state $(5, 1)$ in the partially extracted automata. Thus it appears there is a self-referential connection on input 1 for state $(5, 1)$. **f**: the input 0 is recorded as a self-referential connection on state $(5, 1)$ because with reference to M, the network is actually in state $(2, 1)$, so when it sees 0, it goes to state (2,1), which is represented by state (5,0) in the partial extraction. **g** input 1 is again recorded as a self-referential connection on state $(5, 1)$, equating to a transition into state $(1, 1)$ in M. **h**–**n** as indicated and relatively uninteresting. The bottom right inset shows a table, the first column indexes the figure insets, starting from **c** because this is where the start state is assumed to have been discovered. The second column shows the input sequence used in the extraction steps starting from **c**, the third column shows the states visited in M as a consequence of the input sequence of column two, and the forth column shows the states visited in the extracted structure as a consequence of the input sequence of column two.

This illustrates how an automated procedure can extract an automata with non-deterministic transitions if the clustering mechanism under-represents the states of the induced structure. Furthermore, by comparing insets **a**, and **n** it is not immediately apparent how this can be resolved post-extraction. Obviously state (5,1) is non-deterministic on input 0, and thus perhaps should be split into two states since on this input it goes to states with different outputs. But if state (5,1) were split into two states, and presumably all of its connections duplicated on the child state, then there is still the problem of determining which state (9,0) should connect to on input 1. It certainly should not connect to both, because this would introduce a non-deterministic transition, which would have to be resolved, yet since the new child state and the parent state transition to states with different outputs on input 0, there is no proviso for binding them in some automated disambiguating routine by nature of the way Moore machine minimisation works. It may be the case that it is not possible to determine the induced structure given the configuration shown in inset **n**. There are some possible runtime remedies for this, but none of them were significantly analysed in this research to warrant mentioning them.

There is a problem that can occur in recurrent networks emulating Moore machines, that does not occur in Moore machines. And further, only occurs with recurrent networks that have not completely induced the target Moore machine; an induced state can sometimes have an ambiguously defined output, but **not** as a consequence of the clustering mechanism under-representing the states of the induced automata in its clusters. The problem occurs when the real valued outputs of the trained network are converted to integer outputs suitable for interpretation in the codomain of the function implemented by the target Moore machine. As explained in §4.3, a desired integer output is encoded in binary across the firing times of the output neurons, and in the case of one output neuron, high and low firing times are used to encode logical 1s, and 0s, respectively. This means that in order to disambiguate between output firing times that do not exactly match these high and low values, they must be thresholded about the midpoint of the high and low values. The problem is that sometimes the output of a particular neuron will be very close to the midpoint, and depending on the sequence of inputs preceding the state which bought about the current output, when thresholded, the output will correspond to either a logical 1 or a logical 0, when really it should always correspond to a logical 0 or always correspond to a logical 1 irrespective of the sequence of inputs preceding the state which bought about the current output. This has to be considered when it comes to validating the extracted automata (see next section).

The success of a particular clustering run is thus somewhat stochastic and therefore it is generally a good idea to perform multiple clustering runs and hope that no non-deterministic transitions on outputs are discovered in the extraction procedure. If the situation seems hopeless, the automata can be subjected to a manual extraction. In a manual extraction a human interactively provides each input for the extraction procedure. The computer clusters the current state and returns an identifier; which in the case of clustering networks happens to be the index of the winning neuron. The human can then draw this state on paper. The human then gives the computer an input symbol. The computer simulates

the recurrent network, works out the next $\varpi^{\vec{fire}}$ from $\omega^{\vec{fire}}$, clusters the new $\varpi^{\vec{fire}}$, and then gives the index of the winning neuron to the human, and gives the output of the network to the human. The human can then draw the new state that the computer returned, assuming it is different than any other already on the paper, and draw a transition from the originating state to this new state on the input that was provided. The human can thus iteratively build up the automata on paper, as the computer would in its memory. The advantage of this over automatic extraction is that a human can detect when the result of a transition contradicts with what has already been extracted and written on paper, and consequently assume the existence of another state using some intelligent reasoning given the context of the contradiction, and given this assumed state, perform enumeration of the transitions from the new state in the ordinary manner, recursively dealing with contradictions. Whenever there are contradictions, it can get confusing; multiple states drawn by the human on the paper will be called the same thing by the clustering network, and so to tell them apart, a sequence of inputs must be presented which discriminates between those states on the paper obtaining the same name as the target. It can get marginally complicated, but if the procedure can be formalised, then it can be programmed and consequently automated, this is a topic for further research.

The computer typically prints the $\varpi^{\vec{fire}}$ that is used to obtain the cluster indices at each step, and the non-thresholded output so that the human can observe them, and make note when an output firing time is at the midpoint between low and high firing times as discussed above.

As a sanity check, k-means clustering of $\varpi^{\vec{fire}}$ was tried as an alternative to the clustering mechanism of §3.6. The former appeared to suffer from the same under-representation and over-representation problems as the latter. K-means clustering is significantly faster however because the codebooks can be explicitly computed and stored so that instance classification can be performed later according to the codebook with smallest Euclidean distance from the instance, whereas in spiking neuron network clustering, the codebooks are implicitly stored in the network, and thus to obtain a classification for an instance, the clustering network has to be simulated to obtain the index of the winning neuron. Note that it would be possible to take the trained clustering network, then over some test set, classify each instance according to the index of the winning neuron associated with it, and store this association. Then by averaging the vectors associated with each winning neuron index, a set of codebooks could be obtained and be used in the same manner as is done with K-means. This was not done in practice. Note that the k-means clustering software used was programmed by Dr Peter Tiňo.

5.3 Validation of an Extracted Moore machine

After an MM has been extracted using the clustering mechanism described above, it has to be validated to ensure that the extracted MM is consistent with the induced structure of the RSN. This is achieved by taking the extracted MM and generating a random test set from it. Then, if the structure induced by the trained RSN corresponds to the extracted MM, or a minimal version of it, the network should get zero error over the test set.

6 Experiments

This section documents some successful MM induction experiments using RSN. It shows that RSN can induce persistent structures that enable it to go beyond finite memory.

6.1 TwoState Induction Experiment

6.1.1 TwoState Method

The training set shown in Table-1 for the target MM shown in Figure-11 was concatenated to form a single string, separating each sub-sequence by an implicit reset symbol (See §4.5.4). An RSN (See §3.4) was trained using the SPTTN training paradigm (See §3.5). using a dynamic learning rate (See §4.5.5). A listing of the experiment parameters used is shown in Table-C.12. A dynamic learning rate was used.

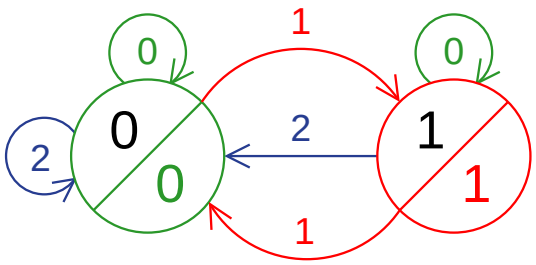


Figure 11: TwoState MM, upper left symbol in each state is a label, and lower right symbol in each state specifies output value.

2222 0000	Reset run
0000 0000	Training sequence 1
0010 0011	Training sequence 2
0110 0100	Training sequence 3
0100 0111	Training sequence 4
1111 1010	Training sequence 5

Table 1: A training set for the MM shown in Figure-11, for the MM induction experiment described in section §6.1

.

6.1.2 TwoState Results

After 1140 epochs the average squared error per symbol per epoch had dropped from initially 82.038571 to 0.888571. The training procedure was continued, and the error varied erratically, eventually reaching a minimum of 0.001786 after 11755 epochs. The training was stopped after 14500 epochs. A graph of the error per epoch is shown in Figure-12, since the error went from 82.038571 to 6.739643 in the first epoch, the first value is not shown, to visually accentuate the remaining differences in error amplitude.

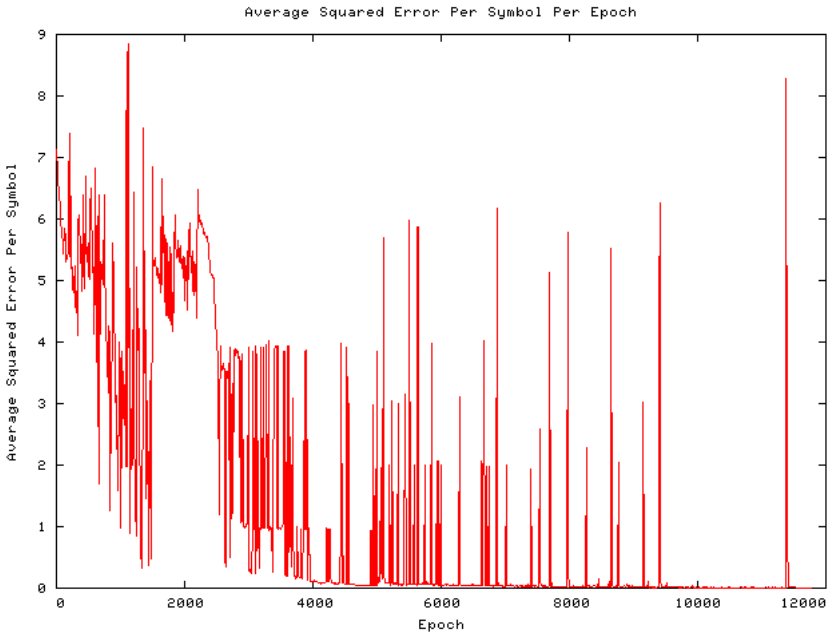


Figure 12: Average Squared Error Per Symbol Per Epoch for the induction training experiment of §6.1. Notice how the error rate is unusually erratic right before reaching the minimum error at the far right of the graph.

The trained RSN was tested over a randomly generated test set $\{0,1,2\}^{10000}$, which was prefixed by 5 implicit reset symbols. The resultant total thresholded error was zero. The average squared error per symbol per epoch was 0.022429.

The induced structure was extracted from the trained network using the methods outlined in §5. It was identical to the original structure shown in Figure-11.

6.1.3 TwoState Validation

Validation was not necessary because the extracted induced MM was identical to the target MM.

6.1.4 TwoState Discussion

This experiment demonstrates that RSN can be trained using gradient descent training to induce the structure of a target MM whose successful induction entails that the trained RSN can go beyond finite memory by emulating the target MM over arbitrary input sequences, as explained in §1.

The error rate is hyper-erratic, this implies either that the error space is hyper-complex, or the learning rate is too high. However, using the words "too high" implies that this is a bad thing, but learning was successful so perhaps it is not so bad. Note that at there is big spike in error shown at the right end of the graph, the minimum error occurred just after this, it is suspected that this spike was caused by a small weight change causing some neurons not to fire, and that then immediately another small change

stopped this, see §7.1, the important thing to note is that behaviour is rarely seen in classical artificial neural networks.

The learning rate used was 1, although this might seem extremely high with reference to classical artificial neural networks, as discussed in §4.5.5, reasonable learning can occur in FFSN when a learning rate of 1 is used, and in fact it turns out that often using a learning rate of 1 obtains a high induction success rate for the simple MM presented here.

6.2 ThreeState Induction Experiment

6.2.1 ThreeState Method

The training set shown in Table-2 for the target MM shown in Figure-13 was concatenated to form a single training sequence, separating each sub-sequence by two implicit reset symbols (See 4.5.4). The training sequence was used to train an RSN using the SPTTN learning paradigm (See §3.5). A listing of the experiment parameters used is shown in Table-C.12. A dynamic learning rate was used.

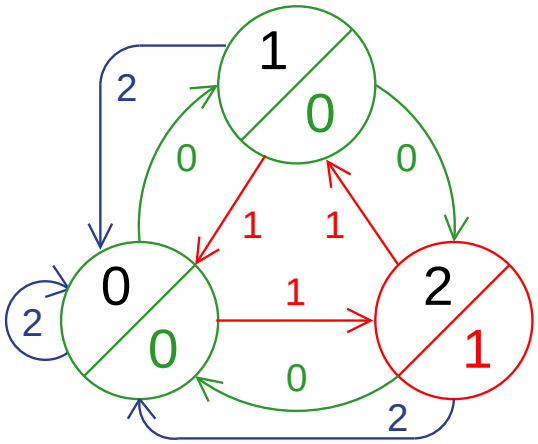


Figure 13: ThreeState target MM, upper left symbol in each state is a label, lower right symbol specifies output value.

222222 000000	Reset run
111111 100100	Training sequence 1
001010 010101	Training sequence 2
110101 010101	Training sequence 3

Table 2: A training set for the MM shown in Figure-13, used in the MM induction experiment described in §6.2

.

6.2.2 ThreeState Results

After 10400 epochs, the average squared error per symbol per epoch reached 0.81222, then started increasing. After 11401 epochs the learning rate, which had decayed to 0.001454, was reset to 1.0, and the network weights reverted to those which previously obtained the lowest error over the training set. At epoch 13890 the error reached a minimum of 0.655556. Training was terminated after 14900 epochs. This is shown graphically in Figure-14.

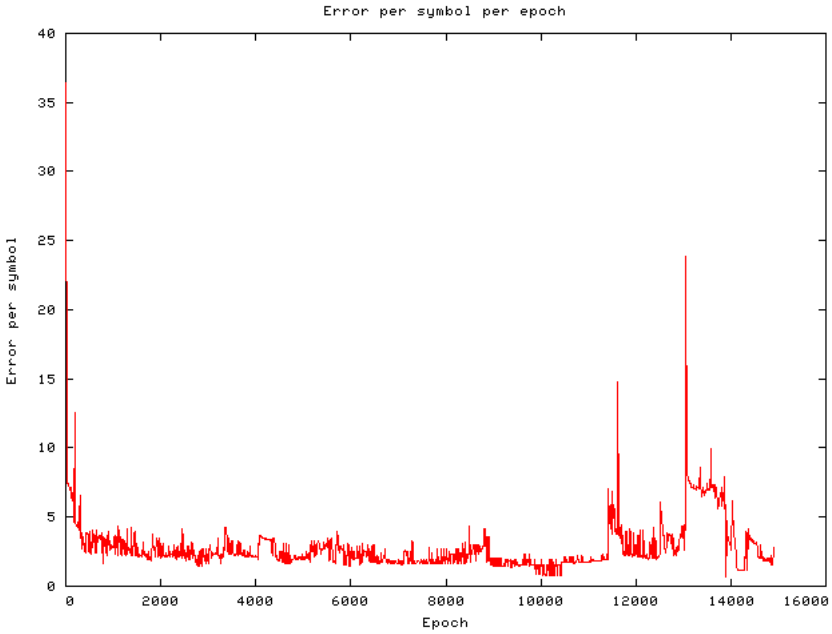


Figure 14: Graph of the average squared error per symbol per epoch for the induction experiment of §6.2. Note the unusually hyper-erratic nature of the error space. Note also that the minimum error occurred after 13890 which is within a region of much erraticity.

Examining the training set error in more detail revealed that the only incorrect symbol after thresholding was the first symbol of the second training sequence.

The trained network was tested over a randomly generated test set $\{0,1,2\}^{10000}$. The test set was prefixed by 5 reset symbols (2). 307/1005 test symbols were incorrectly predicted, indicating an error rate of 0.305473.

K-means clustering was used to obtain ten codebooks from the the recurrent-input firing times $\varpi^{\vec{fire}}$ of the network over the test set, k-means clustering was used over spiking network clustering because of speed advantages. The codebooks were used to interactively extract the network as described in §5. The result of the extraction procedure is shown in Figure-15.

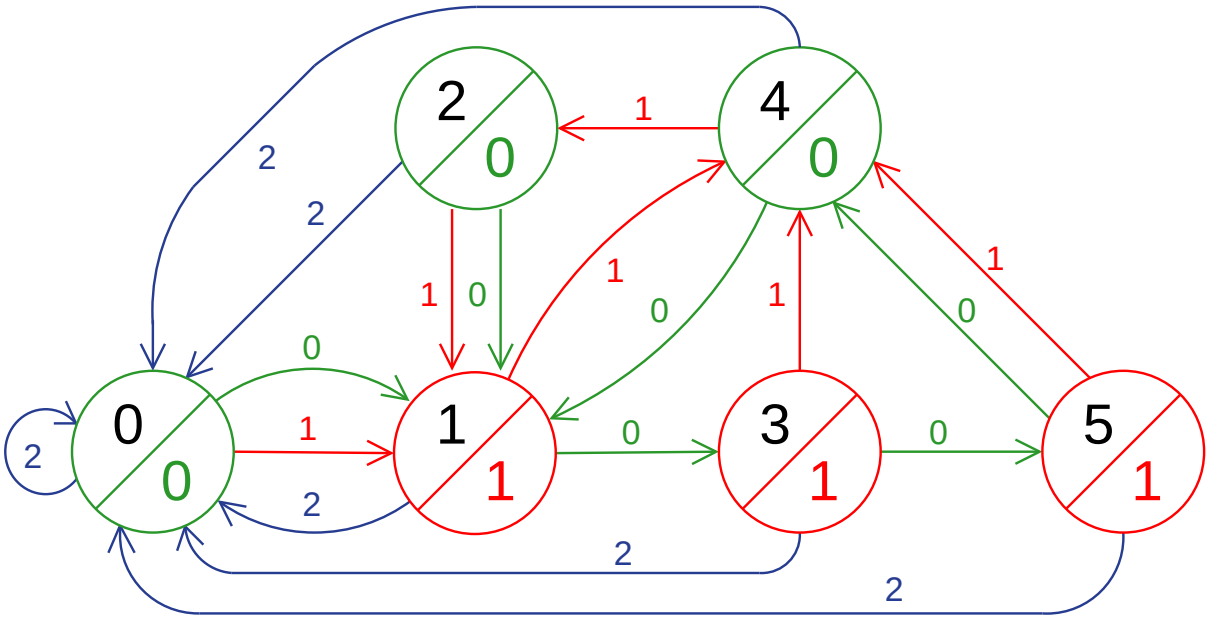


Figure 15: MM extracted from the MM induction experiment of §6.2. Circles represent states. Upper left symbol in each state is a label. The lower right symbol in each state indicates output value.

6.2.3 ThreeState Validation

A randomly generated test set (input output pair consistent with the operation of an MM) of 10000 symbols was generated from the extracted MM, the test set was prefixed with 5 reset symbols. Out of 10005 symbols, 0 were predicted incorrectly, indicating that the extracted structure was indeed that used by the network.

6.2.4 ThreeState Discussion

It is apparent that the inner cycle of the MM shown in Figure-13, that is (111)* has been induced, but the outer cycle, that is (000), has not. The oscillation between states 0 and 1 in the extracted MM corresponds with the oscillation between states 0 and 1 in the original MM. The reset symbol 2 has also been induced correctly. Curiously, a cycle of length 4 has been induced on 0 over the states (1,3,5,4)*, this is interesting because on closer inspection it is evident that the cycle of length 3 on 0 in the original MM, is missing in the training set. The extracted MM is minimal.

It appears as though the training set is critical to induction. Before this experiment was ran, it was ran with much longer training sequences, and the network converged having every output firing at the same time; in the middle between the high and low output values used in the desired output encoding, indicating that the learning process was settling to some average of the many directions in weight space that the many input-output sequences were pulling it. Further experiments have managed to obtain low error rates over training sets including both cycles, however, no RSN has yet been trained that has a generalisation error better than 17%, i.e. that predicts more than 83% of symbols correctly of from randomly generated test string of length 10000.

It appears that if the training set contains too many strings, or they are too long, then this reduces the capacity of the network to learn. It also appears that the outcomes of these experiments are very dependent on the initial weight settings; much variation on the minimum error obtained within a fixed number of epochs has been observed over multiple training runs.

The experiment should be re-executed, but this time including the previously missed-out cycle on 0. Further experiments should be tried with slightly different training sets to compare performance. It is

believed that it should be possible to induce this MM completely given the correct training set.

6.3 Other Experiments

Many other experiments were performed in addition to those described in the previous two sections. Some of them will be mentioned briefly here, for lack of space.

6.3.1 Induction of Cycles

The Moore machines shown in Figure-16 below, were completely induced:

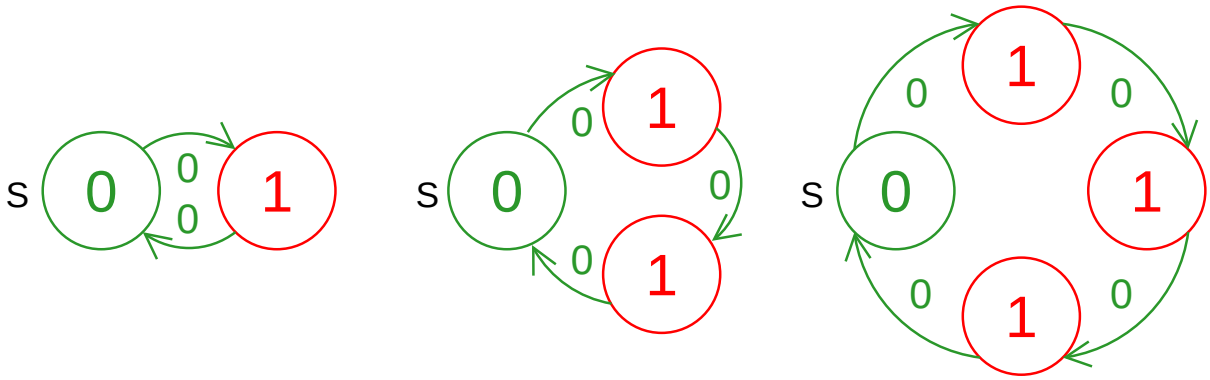


Figure 16: Cyclic Moore machines that were completely induced during the course of the research. Circles represent states. The number in each state represents its output. **s** denotes the start state of each MM.

A similar process was executed to induce each one. First an RSN was trained on one instance of the target cycle, presenting it repeatedly, and then when the error was sufficiently low, two concatenated instances of the cycle were presented, then three concatenated instances, and so on, until arbitrary concatenations of the cycle could be presented whilst maintaining a low error. It was usually the case that after training on just one instance of a cycle an RSN would generalise to arbitrary presentations. It was not possible to induce a cycle of length five in the same manner; all the output firing times went to the midpoint of the high and low output encoding value. Interestingly, equivalent cycle induction is possible upto *at least* length eight in classical recurrent networks (Peter Tiňo, personal communication).

6.3.2 Imposed Start State Experiment

An experiment was carried out to test how much influence imposing a start state has on the learning of a single loop using an RSN. The task was to predict the symbol 1 given repeated occurrences of the symbol 0. This is a very simple task, and it might seem that it does not require that a loop be learned at all, but in practice the recurrent architecture forces $\varpi^{\vec{fire}}$ to converge over time so that the RSN cannot help but learn the function recurrently, obtaining the induction of a single loop. In the first experiment, 100 symbols were presented, with no resets, in the second experiment 100 symbols were presented, presenting an **implicit** reset symbol after every 10 symbols, and in the third experiment 100 symbols were presented, presenting an **explicit** reset symbol after every 10 symbols. See §4.5.4 for information about implicit and explicit reset symbols. The average squared error per symbol over the 100 symbols of each experiment was recorded, the results are shown below:

Reset Type	Error
Implicit reset	0.703374
No reset	0.793140
Explicit reset	2.854510

Where error is average minimum squared error over 100 training runs. These results imply that *for this simple scenario* using an implicit reset can significantly speed up learning, that using no reset is not much worse than using an implicit reset, and that the worst impact on performance comes from using an explicit reset symbol. The extent to which these implications impact the induction of more complicated MM is currently unknown. Future experimentation should address this question.

7 Evaluation

7.1 General Considerations

The experiments described in §6 show the induction of only relatively simple Moore machines (MM), although they were complicated enough to demonstrate an initial research aim; being able to go beyond finite memory. However, even these basic experiments were reasonably hard to repeat when subsequent training runs were executed with different initial network weight settings, implying a strong relationship between the initial weight settings and the outcome of a training run.

The initially high learning rate of 1 used in the experiments probably increases the erraticity of the learning trajectory, but observation of many experiments reveals that the error surface is similarly erratic for learning rates at least an order of magnitude less than this, the error surface can be said to be hyper-erratic, or hyper-complex. One cause for this is when small weight changes cause neurons in the network to "die", that is, small changes alter the potentiation of a neuron so that it no longer fires within the feed-forward simulation phase. A small change in weight can result in a disproportionately large increase in the error rate if a neuron dies as a consequence, a subsequent small weight change can revive the dead neuron again, this behaviour is is probably what accounts for the majority of the large peaks seen in Figure-12 and Figure-15 of §6.

The main problem suffered in this research was an initial inability to obtain a foothold to base future experiments on; much effort was expended obtaining the initial results presented in §6. It was not desirable to construct a sequence of systematic experiments from the beginning since there was little information or evidenced intuition to guide such a pursuit, and given the extremely large state space afforded by the numerous parameter settings attributed to RSN, an exhaustive exploration was computationally infeasible with the available hardware, thus experiments were lead primarily by what worked for CRN, and secondly by intuition and empirical observation. A small amount of experimentation where some parameters were randomly set was also performed. Although finally, an initial foothold has been obtained through the experiments of §6 providing a good starting point for future research.

It is currently not feasible, given the information obtained, to explicitly and quantitatively state the most important determinants to the successful induction of MM in RSN. But an intuition is emerging, it appears that using several short training sequences separated by implicit reset symbols, and a high learning rate that can be adjusted dynamically if necessary, can improve induction success rates.

The experiments implement Δ^{α}, the delay realised by the recurrent-function α, using arithmetic addition. Hence a claim against biological plausibility could be attempted, but as explained in §3.4.2, FFSN can be trained to implement any delay not exceeding the precision afforded by the temporal resolution used by the FFSN in its simulation, and none the delays used in the experiments exceed this temporal resolution, hence, since it is has been shown that FFSN *can* be trained to implement all the delays used in the experiments of this research, the biological implausibility claims disappear and arithmetic can be used for convenience an efficiency.

7.2 Dynamic Learning Rate Considerations

The motivation for using an automated dynamic learning rate, as described in §4.5.5, obtains from observations made whilst controlling the learning rate of training runs manually. For example, successful training can be achieved by using an initially high learning rate, and always saving the minimum-error network weights as training progresses, then when a reasonably low error rate has been obtained, the obtaining network-weights can be recalled and the simulation continued, but at that point lowering the learning rate by an order of magnitude. This is usually quite effective in diminishing to insignificance the remaining error. However, it can happen that reducing the learning rate at that point leads to no further reduction in error, in which case increasing the learning rate again can often bolster the network out of the rut it is in and subsequently result in further error reduction, at the expense of a more erratic learning trajectory, and the risk of damaging any previously learned components due to overly zealous adaptation of the network weights as a consequence of the re-instantiated high learning rate. This kind of

error-rate-considerate manual manipulation of the learning rate is a somewhat ad-hoc procedure, hence the desire obtain a heuristic whose realisation is effective in increasing the success rate of arbitrary training runs. Given such a heuristic, the process could be automated.

Automatic adjustment of the learning rate works optimally when the error surface is characterised by smooth gradients and is continuous, so that a decrease in learning rate brings about an approximately linear reduction in erraticity of the error trajectory, entailing that an appropriately chosen oscillation threshold will efficiently detect and relinquish the learning path from oscillatory attractors by decreasing the learning rate, and appropriately increase the learning rate when a plateau or local minima is detected to escape from it, such that the overall effect is a gradual decrease in the overall error rate through time. The problem is that, presumably as a consequence of the hyper-complex nature of the actual error space, this optimal behaviour does not obtain, the dynamic learning rate does not operate as expected; decreasing the learning rate by an order of magnitude can have no significant effect on the magnitude of the error trajectory erraticity, such that effectively using a dynamic learning rate would not work. But in other cases it appears to be the case that appropriate adjustment of the learning rate would bring about an increase in the subsequent rate of reduction of error:

Reducing the learning rate after a reasonably low error rate has already been obtained by a high learning rate, can result in the remaining error being diminished quite quickly, where otherwise, if the learning rate had been left fixed, no significant decrease in error would have been obtained. This is not a speculation, but an empirically evidenced observation; the network weights obtaining the minimum error so far can be saved during training so that at any time, they can be recalled, thus multiple realities can be played out from a fixpoint by changing the learning parameters, recalling the saved weights, and then continuing training. For this to make any sense, the reality which does not result in an error decrease has to occur first, otherwise the original fixpoint weights would be replaced by those responsible for a new, lower error. This is not a problem because it is in precisely those instances where the error has not decreased since a fixpoint when it becomes desirable to recall the best weights and try a parameter adjustment because it is already evident that the current trajectory has not obtained a better error. Some observations are listed below:

1. The learning rate of an RSN was fixed at a constant high value and after the error reached a minimum, tens of thousands of epochs passed without any further decrease in error, but when the high learning rate was changed to a low learning rate, the best network weights reloaded, and training continued, the result was that the error diminished significantly over the next few thousand epochs resulting in successful induction of the target Moore machine.

2. The learning rate of an RSN was fixed at a constant low value, and after the error reached a minimum, the learning rate was lowered as above, and left for thousands of epochs but no decrease in error was observed, but when the learning rate was reset to a high learning rate, and the best network weights reloaded, the result was a significant decrease in error rate within approximately the same time period that the former had failed to obtain a lower learning rate in, and successful induction of the target Moore machine.

3. A high error rate was used from the outset of an RSN training regime, without automatic dynamic adjustment or manual alteration for the entire training duration, the result was a successful reduction in error, stabilisation of the error trajectory, and induction of the target Moore machine.

4. A low learning rate was used from the outset of a training regime, without automatic dynamic adjustment or manual alteration, the result was that the error rate remained high compared to the other listed observations, for the duration of the training regime, and no structure was induced. In fact, the output firing times settled on the midpoints between the high and low desired values, indicating that the network had learned nothing.

Observations obtaining the same behaviour as those described above were observed on a significant number of occasions, indicating that the behaviour they explain is stereotypical. From the first observation it appears that switching from a high learning rate to a low learning rate after a reasonably low error minimum has already been obtained can result in improved induction success, but from the second observation it is apparent that this does not always work. The third observation indicates that using a high

learning rate and not decreasing can also be a successful strategy. And the final observation suggests that using a low learning rate for the entire duration of an RSN training regime does not appear to obtain success in RSN training. Taking into consideration the frequency of these observations it appears as though a large learning rate is needed to find initial structures, but that that sometimes, with a high enough frequency of occurrence to suggest correlation, substituting a small learning rate is more effective in refining those structures and obtaining stability than maintaining a high learning rate, that using a small learning rate from the offset is not effective, and that sometimes it is better to leave the learning rate high. This reasoning entails that it is beneficial to lower the learning rate when the partially induced structure has reached some critical state such that there is a learning path from the current point in error space to somewhere sufficiently close to where the the target structure would obtain. Creating a heuristic that predicts that such a path will obtain if the error rate is reduced at a particular point in the training, has not yet been achieved because the factors on which the success of such learning rate modifications depend, have not yet been sufficiently qualified through analysis and observation. The observations outlined above were formulated from what was observed over several months of training RSN, thus should be verified using a formal experimental procedure.

7.3 Future Exploration

There are numerous possible avenues for future experimentation, these are listed briefly below.

1. The FFSN simulation mechanism described in §3.3 increments time in discrete steps, thus it will often either under-estimate or over-estimate the actual firing time of a neuron. An efficiency optimisation and an improvement in the estimation of neuron firing time could be obtained by using interpolation to approximate the firing time to more precision. Although, it could be argued that it is better to not interpolate and use a temporal resolution in the adaptation phase which is consistent with the temporal resolution in the feed-forward simulation phase in case using different temporal resolutions somehow alters the effectiveness of their interaction. This should be investigated further.

2. When clustering state-input firing times, it was observed that spiking network clustering would often under-represent clusters, and this was also observed for standard k-means clustering. The spiking clustering mechanism of §3.6 does not use a saturation function when adjusting network weights, and does not use slow self-inhibition at the output neurons during training as suggested in [26], further experimentation should ascertain if any improvement in the discriminatory capacity of the network obtains from using a saturation function, or using slow self-inhibition at the output neurons. Furthermore, whenever the clustering mechanism was used in the experiments thus far performed, w^{min} was always set to zero, thus the effect of varying w^{min} should also be assessed. Finally, it might also be beneficial to decrease the width of the learning function through time as is done in the classical SOM learning algorithm. If none of these mechanisms obtain an improvement in discriminatory capacity, then a survey should be affected to see if improvement can be obtained using an unsupervised training paradigm different from the k-means and CN clustering paradigms applied thus far in this research, which both demonstrated similar problems.

3. It is claimed in §4.3 that the number of ι^s should be approximately equal to the number of ϖ to balance the presentation of state and symbolic input information. However, in the experiments of §6, forgetting this claim when setting up the configuration, and paying insufficient attention resulted in meant that twice as many ι^s were used than ϖ. Thus further experimentation should be performed to ascertain if doing this had any significant impact on induction success rate. However, note that upon realising this, the number of ι^s was been set equal to the number of ϖ and observation thus far indicates that using an even number of input neurons for each bit, makes no discernible difference to the induction success rate of the experiments in §6.

4. The experiments of §6 use only one output neuron, and thus are limited to only two output symbols, because it has been demonstrated that using more than one output neuron per bit within SPTTN training seriously disrupts the network's capacity to learn. Yet, if SPTTN training is performed without the contraction phase (See §3.5), then learning with ten output neurons per bit, and sixteen different output symbols has been shown easily achievable. Thus this leads to a questioning

of the way recurrency is obtained in SPTTN. Future experimentation should reconsider the network architecture, or at least ascertain why their is such a distinction between SPTTN training without the contraction phase, and with the contraction phase. So that it may not be misconstrued, let it be noted that SPTTN training without the contraction phase is nothing more than FFSN training in parallel over many FFSN, thus success in SPTTN training without the contraction phase does not imply anything about the ability of SPTTN training to induce MM . Note also that the same problem attributed to using multiple output neurons also occurs with standard RSN training, which obtains more evidence suggesting that there the recurrent mechanism should be analysed in more detail. This is quite important because by being limited to only one output neuron, there is a limit on the freedom to choose arbitrary target MM.

5. In the spiking network clustering mechanism discussed in §3.6 weight bounds are imposed following suggestions made in [26], and observation indicates that this is essential for effective unsupervised learning using CN. Future experimentation should assess whether analogous use of weight bounds in the supervised learning of FFSN and RSN obtains any benefit.

6. In §5.2 some problems associated with clustering using spiking neurons, and the consequent problems with automated and manual extraction of induced structures are discussed. Future work should investigate the construction of more effective automated extraction procedures that can deal with the problems discussed.

7. The test sets used to test trained RSN (§4.4), and used to extract the states of an induced structures from these RSN (§5.1), are generated randomly from the set of input symbols used by the target MM. However, using completely random test sequences will not exercise all areas of a complicated MM equally often due to inherent bias in the structure of the MM, for example a cycle of length five on a single input symbol is relatively unlikely to be presented three times in a row using an entirely random test set generation mechanism, but it is nevertheless necessary to exercise this cycle when testing for generality otherwise a false measure of generality might obtain from testing, thus future work should create a mechanism to generate random test sequences which, in addition to containing entirely random sequences, have subsequences that are biased towards the structure of the target MM so that its structure is sufficiently exhausted during testing of the trained RSN. Note that the target MM used in the experiments of §6 are sufficiently simple such that the purely random test sets used when testing their generality suffices to exercise their structures.

8. The networks used in the induction experiments of §6 only have reference neurons in the input layer. Effort should be expended to assess the importance of reference neurons in the success of training, as it is currently not entirely clear what benefit they obtain, if any. If reference neurons are found to be beneficial, and the reason can be explained then, experiments should also be performed to determine whether including reference neurons in subsequent layers obtains any additional benefit. Note that the reason reference neurons have been used in this research, without understanding why, was because they were used in the majority of the reference material, although from recollection, this material didn't explain why either, but it was subsequently forgotten that this was the case until now, because such considerations were overshadowed by seemingly more important ones.

9. The weight initialisation and parameter setting methods described in §4.5.5 demonstrate a discordance with classical artificial neuron networks. This can be explained partially by the need to scale PSPs sufficiently so that neurons actually fire, however it does not explain Spikeprop's apparent reduction in learning capability when the threshold and weights are scaled down to the ranges used by classical neural networks, hence future work should attempt to determine the reason for this.

10. Intuitively it would seem that the network obtains a capacity to correctly predict the next state, by virtue of the target outputs at each simulation epoch being the output associated with the next state, which forces the recurrent-from neurons to obtain some representation of the next state if the outputs are learned correctly because their firing times are determinants of the output neuron firing times. Thus it would be interesting to compare the training performance when the current state outputs are used as targets, as apposed to next state targets. It is hypothesised that successful training will be unobtainable using this method, hence this hypothesis should be tested in future work.

11. The start times ϖ^{start} are imposed on an RSN during training at the start of each presented training sequence, future investigation should assess the extent that imposing different kinds of start times impacts on the success rate of induction training, a preliminary experiment has already been performed (See §6.3.2), but this should be augmented with more sophisticated, extensive, revealing experiments, and in depth theoretical analysis.

12. As first mentioned in §2.1, the spiking neuron network model used in this research binds response function types at the neuron layer, where as the software (See requirements 1.3.3.1 and 1.3.3.2 of §B.1) allows response function types to be bound at either the synaptic level or at the neuron level. All of the experiments of §6 bind the response type at the neuron level, thus future experimentation should assess the effect that binding and distributing inhibition at the synaptic level has on the success rate of induction training.

13. The basis for derivation of the Spikeprop learning rule discussed in §2.3 is sum squared error, a possible route for future research could be to experiment with learning equations derived from different error functions such as cross entropy.

14. When a neuron stops firing because the weights positively potentiating its activation are too small, there is a possibility that it will never fire again because the weights will not be updated until that neuron fires again, thus the only way to get the neuron to fire again, is if the firing times of predecessor neurons to shift in a direction that entails an increase in the potentiation of the neuron that has ceased firing beyond its firing threshold. Neurons can also stop firing as a side effect of updates to neurons not directly concerned with the affected neuron. This arises as a consequence of the model being used, but is intuitively not what is really wanted, instead it is desired that the network recognise that a neuron not firing is equivalent to it firing extremely late, i.e. infinitely late. Future work should investigate the effect on induction success rate obtained by assigning a large firing time to non-firing neurons so that it appears as if they have fired too late so that the weights mediating its firing time are altered during the adaptation phase in such a manner that eventually the neuron is bought back to life. It is worth noting that doing this would entail being able to justify it biologically and theoretically, and in some cases it could be that the non-firing of a neuron is beneficial to the global error, and a health consequence of the gradient decent procedure, therefore these considerations should be analysed in more depth.

15. In order to obtain a better intuition about the operation of spiking networks, future work should create a system to visualise in realtime the transmission and combination of PSP across synapses and at neurons. Additionally it might be of benefit to explore a very thinly covering partial-exhaustion of the parameter state space a goal of optimising the induction success rate of simple MM, under an assumption, that such optimisations will generalise to the induction of different and more complicated MM.

8 Discussion

Research performed with classical recurrent neural networks (CRN), that is, second generation networks as discussed in §1, has demonstrated their excellent grammatical inference capabilities through the successful induction of relatively complex automata and has even been able to demonstrate prediction of chaotic sequences. Induction of FSM using CRN has been used as an exercise for students at Aston University (Peter Tiňo, personal communication), this implies that the notion of FSM induction using CRN is reasonably well established. It was therefore thought initially, that similar research aims applied to the domain of spiking neuron networks with recurrent connections (RSN), would demonstrate an at least basic grammatical inference capability without significant additional effort when compared to the effort required to achieve comparable results in the CRN domain. It turned out however, that obtaining results that are somewhat trivial to obtain with CRN, was a seemingly non-trivial task using RSN. In retrospect, the original speculation was unfounded due to the way that information is processed within CRN and RSN being fundamentally quite different, and the fact that the former applies continuous activation function to the activation of affected neurons, whereas the latter applies a threshold function at the neuron level as well as continuous response functions at the connection level. The following few paragraphs discuss several of the important distinctions between RSN and CSN that could explain the increased training difficulty observed.

In CRN, arbitrarily sized inputs from $\Re$ can be used directly as the activations of input neurons because of the squashing action of the sigmoid function at affected hidden neurons, whereas with RSN, the inherent temporal dynamics must be accounted for; inputs and outputs must be temporally encoded (See §4.3) and the design of the recurrent architecture must be considerate of the temporal relationships between sequential simulation epochs (See §4.5.2 and §4.5.3).

In CRN, information from the recurrent-from layer can be fed directly back into the input layer at each simulation epoch without losing any information, so that symbolic input information and state information are effectively combined, but with RSN, simply feeding the firing times of the recurrent-from neurons back into the input layer across the state-input neurons is not acceptable, because of a biological plausibility constraint imposed in §4.5.2 concerning the intra-input interval Υ. Thus, appropriate temporal translation of recurrently derived state information with respect to symbolic input information is necessary so that these informations obtain a temporal proximity at the beginning of each simulation epoch, however, if the temporal translation coefficient applied to the recurrent-from neuron firing times under or over estimates the amount of translation required, then state information presented at the start of each simulation epoch can gradually shift out of phase with symbolic input information, resulting in gradual information loss with respect to state input information, and culminating in an attenuation of the network's capability to integrate this information with symbolic input information, consequently degenerating learning potential.

A neuron ceases to fire within the feed-forward simulation phase when the superposition of the PSP which determine its membrane potential no longer exceed its threshold within the duration of this phase. This can occur either as a side effect of the modification of the firing times of other neurons, or can occur directly as an effect of attempting to change the firing time of the neuron itself. The latter obtains more interest here than the former because the former is ultimately a consequence of the latter being applied to other neurons, and hence is more fundamental than the former.

In the case where output neurons do not fire, it becomes hard to quantify what the error associated with said neurons should be; usually the error associated with an output neuron is the absolute difference between its actual firing time and its desired firing time, but in the case where a neuron does not fire, there is no actual firing time, and so it is difficult to define a reasonable measure of error in such a case. Firing time difference is also used when calculating the output neuron deltas during the Spikeprop adaptation phase; relative firing times of neurons are used in the calculation of both the output and hidden layer deltas. This means that if a neuron does not fire, then it has to be ignored in the adaptation phase when calculating deltas, because as explained above, the difference in firing times between a neuron that does not fire and a neuron that does fire, is undefined. This illustrates a problem associated with RSN that does not occur in CSN; in CSN there is no equivalent to a neuron not firing, because the activation of the neuron is governed by a differentiable function which always produces an output regardless of the input,

as apposed to a threshold function which either produces an output or does not produce an output. This entails that even for very low neuron activation, information about how every neuron in a CSN contributes to its output activations is available for use in the adaptation phase, so that appropriate adjustments to mediating weights can be made. Whereas when a neuron does not fire during feed-forward simulation of an RSN, the contribution of this neuron in determining the firing times of the output neurons, i.e to the overall error, has to be ignored during subsequent adaptation using Spikeprop because there is missing, or otherwise undefined information about the relation between the non-firing neuron to the target output firing times, i.e the appropriate error derivatives cannot be calculated.

If the firing time of a neuron is to be adjusted, then this is because the deltas calculated in the adaptation phase imply that doing so will result in a better approximation of some desired-firing-time vector over the output neurons. It is *not* necessarily the case that increasing or decreasing the firing time will result in respectively an increase or decrease in the potentiation affected by this neuron at successor neurons, because the affecting function transmitted by a neuron is non-linear by virtue of each neuron obtaining multiple axonal delays which are mediated by separate weights. This is also true for sigmoidal CSN because the mediating weights can be negative.

In a classical neural network CRN, the sum of the weights on connections afferent to a hidden layer neuron, i.e the activation, is perturbed by a usually constant, and usually sigmoidal activation function. The result is then scaled by the weighted efferent connections of this neuron during the simulation of the CRN to determine the activation afferent to the successors due to it. The value now associated with each efferent connection is equivalent to the value that would be obtained by scaling the original activation function of this neuron by the efferent connection efficacy to obtain a new function, and then applying this new function to the aggregate activation afferent to the original neuron. Hence each connection efferent to the original neuron, and in general each connection in a CRN, has associated with it a function.

The same is true for FFSN, but the difference is that the functions implemented by the connections of a CRN depend only on the network weights, that is, irrespective of the inputs, feeding the same absolute value into a particular function associated with a connection will always produce the same result. However in an FFSN, the firing times of neurons heterogeneously translate the functions implemented by each connection through time, either in a forward or reverse direction, such that feeding the same absolute value into one of these functions will solicit different responses, depending on the firing times of the input neurons. That is, with respect to absolute responses, the functions are themselves functions of the the input firing times. This is a subtle distinction because each of the functions associated with the efferent connections of a neuron belonging to an FFSN *will* solicit the same response individually, irrespective of the firing times of the input neurons, when fed a constant value *relative to the firing time of the presynaptic neuron.*

The form of the functions used by an FFSN are more complicated than those used by a standard CSN because each connection actually consists of m synapses, each with an associated axonal delay. So whereas the functions used by the CSN attributed to each connection are all stereotypes of some activation function such as the sigmoidal activation function $f(x) = 1/(1 + \exp(-x/\delta))$, the functions used by an FFSN per connection are each the superposition of m PSP response functions $\epsilon_{ij}^k(t) = ij_{type}^k \cdot ((t - t_i^a)/\tau) \cdot exp(1 - ((t_i^a)/\tau)) \cdot \mathcal{H}(t - d_{ij}^k)$. This is complicated further because the functions are defined in terms of the firing time t_i^a of a presynaptic neuron i, thus until i fires, t_i^a is undefined hence the output of the function $\epsilon_{ij}^k(t)$ is undefined, thus if i never fires, all the outgoing connections of i are undefined and so neuron i directly contributes nothing to the firing times of other neurons in the network during the feed-forward simulation phase and has to be ignored in the adaptation phase, as discussed above. To clarify, the functions associated with the outgoing connections of neurons in an FFSN are not fully defined until those neuron fire, whereas the functions associated with the outgoing connections of neurons in a CSN are fully defined by the weights before any input is presented. The result of this is that ultimately spiking neuron networks really are temporally dependent, and this makes them much harder to analyse, debug, and train.

9 Conclusion

The aim of this research was to gain an initial insight into the finite state machine (FSM) (§1) induction capabilities of feed-forward networks of spiking neurons (§2.2,§3.3) with recurrent connections (RSN) (§3.4) because FSM are considered to be good models for computations on time series whereas it is still unknown how biological networks of neurons perform computation on time series. The research focused on a specific type of FSM called a Moore machine (§4.1) and a specific goal was to demonstrate that the induction of persistent structures, that allowed the trained RSN to go beyond finite memory (§4.2). A special form of training called Spikeprop-through-time (§3.5) was used in the induction experiments (§6), and induced structures were extracted (§5) using k-means and spiking network clustering (§3.6). The experiments (§6) achieved the research aim; Moore machine structures that go beyond finite memory have been shown inducible in feed-forward networks of spiking neurons with recurrent connections.

Although the original aim has been achieved, the many points raised in §7, and the comparison of CNN and RSN operation in §8 shows that many questions about the operation of RSN remain unanswered leaving plenty of scope for future work.

A Algorithms

Each algorithm has an associated table specifying its requirements, and the type of those requirements. There are three requirement types: Explicit means that the argument is a direct argument of the algorithm under consideration. Implicit, means that it is is implicitly specified by being part of an explicit requirement, for example the set of all neurons V is inherently part of a feed forward network of spiking neurons Ξ. Implicit requirements provide extra information that might be useful in understanding the algorithms outside of the context of the main text. Context dependent requirements are requirements of the algorithms called within the algorithm under consideration, some of them might be passed explicitly to the algorithm under consideration, and some of them might be generated within the algorithm under consideration, check the algorithm header argument list if in doubt. It is also worth noting that a vector of desired firing times for a set of neurons $x^{desire\vec{d}fire}$ is a direct reference to the actual firing times $t_q^a \in x^{desire\vec{d}fire}$ of the neurons contained in the set. Table-A shows some common notational conventions used throughout the algorithms.

Notation	Explanation
$\Xi^{\Xi_{\circlearrowleft}}$	A Spikeprop through time network (SPTTN)
$\Xi_{\circlearrowleft}$	A recurrent spiking network (RSN)
$\varpi \in \Xi_{\circlearrowleft}$	Set of neurons in recurrent-input layer of $\Xi_{\circlearrowleft}$
$\omega \in \Xi_{\circlearrowleft}$	Set of neurons in recurrent-from layer of $\Xi_{\circlearrowleft}$
Ξ	A feed-forward spiking network (FFSN)
$V \in \Xi$	All neurons (graph vertices) in Ξ
$\iota \in \Xi$	Set of neurons in input layer of Ξ
$h \in \Xi$	Set of neurons in hidden layers of Ξ
$o \in \Xi$	Set of neurons in output layer of Ξ
Ξ_{layers}	The lth layer of Ξ
Ξ_l	The set of neurons in the lth layer of Ξ
Γ_q	The predecessors of neuron q
q^ρ	The membrane potential of neuron q
t_q^a	The actual firing time of neuron q
δ_q	The Spikeprop delta of neuron q
Υ	An intra-input difference
t^{start}	A simulation start time
t^{len}	A simulation duration
t^{end}	The end time of a simulation
t^{inc}	A simulation time increment
α	A recurrent function
$n \leftarrow a$	n is assigned value a
$x^{f\vec{ire}}$	A vector of firing times for the set of neurons n
$x^{desire\vec{d}fire}$	A vector of desired firing times for the set of neurons n
w^{min}	Lower weight bound
w^{max}	Upper weight bound

Table 3: Reminder of notation used in algorithms

Requirement	Type
A forward network of spiking neurons $net \leftarrow \Xi$	Explicit, provided
Or a recurrent network $net \leftarrow \Xi_{\circlearrowleft}$	Explicit, provided
The set of all neurons V	Implicit, $\in net$
Another name for V; $net_{layers} = \{net_1 \dots net_{\lvert net_{layers} \rvert}\}$	Implicit, $\in net$
A partition of net_{layers} into input ι, hidden h, and output o neuron sets	Implicit, $\in net$
A neuron firing threshold ϑ	Explicit, provided
A simulation start time t^{start}	Explicit, provided
A time increment t^{inc}	Explicit, provided
A simulation duration t^{len}	Explicit, provided

Table 4: Input requirements for $FFSNSimulate$ [Algorithm-1]

Algorithm 1 $FFSNSimulate(\{\Xi, \Xi_{\circlearrowleft}\}, \vartheta, t^{start}, t^{inc}, t^{len})$

$(\forall q \in (h \cup o)) \quad (t_q^a \leftarrow -1)$
$t \leftarrow t^{start}$
$t^{end} \leftarrow t^{start} + t^{len}$
while $(\exists j \in o : t_j^a < 0) \wedge (t < t^{end})$ **do**
  $(\forall q \in V) \quad (q^\rho \leftarrow 0)$
  for $l \in \{1 \dots |\Xi_{layers}|\}$ **do**
    for $j \in \{1 \dots |l|\}$ **do**
      if $t_j^a < 0$ **then**
        for $i \in \Gamma_j$ **do**
          if $t_i^a \geq 0$ **then**
            $(\forall k \in \{1 \cdot m\}) \quad (j^\rho \leftarrow j^\rho + \epsilon_{ij}^k (t - d_{ij}^k - t_i^a) \cdot w_{ij}^k)$ [Equation-5]
          end if
        end for
        if $(j^\rho > \vartheta) \wedge (t_a^j \equiv -1)$ **then**
          $t_j^a \leftarrow t$
        end if
      end if
    end for
  end for
  $t \leftarrow t + t^{inc}$
end while

Requirement	Type
A forward network of spiking neurons $net \leftarrow \Xi$	Explicit, provided
Or a recurrent network $net \leftarrow \Xi_{\circlearrowleft}$	Explicit, provided
The set of all neurons $V \in net$	Implicit, $\in net$
Another name for V; $net_{layers} = \{net_1 \dots net_{\lvert net_{layers} \rvert}\}$	Implicit, $\in net$
A partition of net_{layers} into input ι, hidden h, and output o neuron sets	Implicit, $\in net$
Desired output firing times $o^{\vec{desired}fire}$	Explicit, provided

Table 5: Input requirements for $FFSNAdapt$ [Algorithm-2]

Algorithm 2 $FFSNAdapt(\{\Xi, \Xi_{circlearrowright}\}, o^{desiredfire})$

$(\forall q \in o) \quad (\delta_q \leftarrow delta^{out}(q))$ [Equation-12]
for $l \in \{(|\Xi_{layers}| - 1) \ldots 2\}$ **do**
  $(\forall q \in \Xi.l) \quad (\delta_q \leftarrow delta(q))$ [Equation-13]
end for
for $l \in \{(|\Xi_{layers}| - 1) \ldots 1\}$ **do**
  for $i \in \Xi_l$ **do**
    for $j \in \Xi_{(l+1)}$ **do**
      $(\forall k \in \{1 \ldots m\}) \quad (w_{ij}^k \leftarrow WeightUpdate(i, j, k))$ [Equation-16]
    end for
  end for
end for

Requirement	Type
A set of inputs I	Explicit, provided
A set of corresponding outputs O	Explicit, provided
A temporal encoder $\mathcal{T}_{in}^{encode}$ for I	Explicit, provided
A temporal encoder $\mathcal{T}_{out}^{encode}$ for O	Explicit, provided
An error function $error : \langle o^{desired}, o \rangle \rightarrow \Re^+$	Explicit, provided
A lower error bound ϵ^{bound}	Explicit, provided
A learning rate η	Explicit, provided
$FFSNSimulate$ requirements of Table-4	Context dependent
$FFSNAdapt$ requirements of Table-5	Context dependent

Table 6: Input requirements for $FFSNTrain$ [Algorithm-3]

Algorithm 3 $FFSNTrain(\Xi, \mathcal{I}, \mathcal{O}, \mathcal{T}_{in}^{encode}, \mathcal{T}_{out}^{encode}, error, \epsilon^{bound}, \vartheta, t^{start}, t^{inc}, t^{len}$

$\epsilon \leftarrow \epsilon^{bound} + 1$
while $(\epsilon > \epsilon^{bound})$ **do**
  $\epsilon \leftarrow 0$
  for $(i \in \{1 \ldots |\mathcal{I}|)$ **do**
    $t^{start} \leftarrow 0$
    $\iota \leftarrow \mathcal{T}_{in}^{encode}(\mathcal{I}_\rangle)$
    $FFSNSimulate(\Xi, \vartheta, t^{start}, t^{inc}, t^{len})$ [Algorithm-1]
    $o^{desiredfire} \leftarrow \mathcal{T}_{y}^{encode}(\mathcal{O}_i)$
    $e \leftarrow e + error(o, o^{desiredfire})$
    $\Xi \leftarrow FFSNAdapt(\Xi, o^{desiredfire})$ [Algorithm-2]
  end for
end while

Requirement	Type
A recurrent network $net \leftarrow \Xi_{\circlearrowleft}$	Explicit, provided
A forward network of spiking neurons Ξ	Implicit, $\in net$
A set of recurrent input neurons $\varpi \subset \iota$	Implicit, $\in net$
A set of recurrent-from neurons ω	Implicit, $\in net$
A vector of recurrent input start times ϖ^{start}	Implicit, $\in net$
A recurrent-function α	Explicit, provided
An intra-input difference Υ	Explicit, provided
$FFSNTrain$ requirements of Table-6	Context dependent

Table 7: Input requirements for $RSNTrain$ [Algorithm-4]

Algorithm 4 $RSNTrain(\Xi_{\circlearrowleft}, \mathcal{I}, \mathcal{O}, \mathcal{T}_{in}^{encode}, \mathcal{T}_{out}^{encode}, error, \epsilon^{bound}, \vartheta, t^{start}, t^{inc}, \Upsilon, \alpha, t^{len})$

while $\epsilon > \epsilon^{bound}$ **do**
  $\epsilon \leftarrow 0$
  $n \leftarrow 0$
  for $i \in \{1 \dots |\mathcal{I}|\}$ **do**
    $(\iota - \varpi)^{\vec{fire}} \leftarrow \mathcal{T}_{in}^{encode}(\mathcal{I}_i)$
    if $n \equiv 0$ **then**
      $\varpi^{\vec{fire}} \leftarrow \varpi^{start}$
    else
      $\varpi^{\vec{fire}} \leftarrow \alpha(\omega_{end(n-1)}^{\vec{fire}})$
    end if
    $t^{start} \leftarrow (n \cdot \Upsilon)$
    $FFSNSimulate(\Xi^{\circlearrowleft})$ [Algorithm-1]
    $o^{\vec{desiredfire}} \leftarrow \mathcal{T}_{out}^{encode}(\mathcal{O}_i)$
    $e \leftarrow e + error(o, o^{\vec{desiredfire}})$
    $FFSNAdapt(\Xi, o^{\vec{desiredfire}})$ [Algorithm-2]
    $n \leftarrow n + 1$
  end for
end while

Requirement	Type
A Spikeprop through time network $net \leftarrow \Xi^{\Xi\circlearrowleft}$	Explicit, provided
Containing a base recurrent network $\Xi_{\circlearrowleft}$	Implicit, $\in net$
Containing a feed forward network Ξ	Implicit, $\in \Xi_{\circlearrowleft}$
Variable length *copies* of $\Xi_{\circlearrowleft}$	Implicit, $\in net$, depends on input length
A sequence of inputs I	Explicit, provided
Recurrent input start times ϖ^{start}	Implicit, $\in net$
Recurrent function α	Explicit, provided
A function $MakeCopy$ that clones a $\Xi_{\circlearrowleft}$	Implicit, assumed to exist
$FFSNSimulate$ requirements of Table-4	Context dependent

Table 8: Input requirements for Algorithm-5

Algorithm 5 $SPTTNSimulate(\Xi^{\Xi\circlearrowleft}, \mathcal{I}, \alpha, \vartheta, t^{start}, t^{inc}, \Upsilon, t^{len})$

$(\forall q \in \{1 \dots |\mathcal{I}|\})$ $(copies[q] \leftarrow MakeCopy(\Xi^{\circlearrowleft})$
$n \leftarrow 0$
for $q \in \{1 \dots |\mathcal{I}|\}$ **do**
  $t^{start} \leftarrow (q-1) \cdot \Upsilon$
  $((\iota - \varpi) \in copies[q])^{\vec{fire}} \leftarrow \mathcal{T}_{in}^{encode}(\mathcal{I}_q)$
  $(\forall p \in \{1 \dots |((\iota - \varpi) \in copies[q])|\})$ $(t_p^a \leftarrow t_p^a + t^{start})$
  if $n \equiv 0$ **then**
    $(\varpi^{\vec{fire}} \in copies[q]) \leftarrow \varpi^{start}$
  else
    $(\varpi^{\vec{fire}} \in copies[q]) \leftarrow \alpha(\omega_{end(n-1)}^{\vec{fire}})$
  end if
  $FFSNSimulate(copies[q], \vartheta, t^{start}, t^{inc}, t^{len})$ [Algorithm-1]
end for

Requirement	Type
An SPTTN network $net \leftarrow \Xi^{\Xi\circlearrowleft}$	Explicit, provided
$o^{\vec{desiredfire}}$ for all $copies \in \Xi^{\Xi\circlearrowleft}$	Explicit, provided
A learning rate η	Explicit, provided

Table 9: Input requirements for Algorithm-6

Algorithm 6 $SPTTNAdapt(\Xi^{\Xi_{\circlearrowleft}}, \Xi_c^{\Xi_{\circlearrowleft}} opies^{desir\vec{e}dfire}, \eta)$

for $c \in \{|copies| \dots 1\}$ **do**
  $(\forall q \in copies[c]_o) \quad (\delta_q \leftarrow delta^{out}(q))$ [Equation-12]
  for $l \in copies[c]_{layers}$ **do**
    if $(copies[c]_l \equiv copies[c]_\omega) \wedge (l \neq |copies|)$ **then**
      for $q \in copies[c]_l$ **do**
        Subtract Δ^{ax} from $copies[c{+}1]_{h_0}$ and use these values as the firing times for extra predecessors when calculating the delta for neuron q using Equation-13.
      end for
    else
      $(\forall q \in copies[c]_l) \quad (\delta_q \leftarrow delta(q))$ [Equation-13.]
    end if
  end for
end for
for $c \in \{|copies| \dots 1\}$ **do**
  for $l \in \{|(copies[c]_{layers}| - 1) \dots 1\}$ **do**
    for $i \in copies[c]_l$ **do**
      for $j \in copies[c]_{(l+1)}$ **do**
        $(\forall k \in \{1 \dots m\}) \quad ((w_{ij}^k \in \Xi_{\circlearrowleft}) \leftarrow WeightUpdate(i, j, k))$ [Equation-16]
      end for
    end for
  end for
end for

Requirement	Type
An SPTTN network $\Xi^{\Xi_{\circlearrowleft}}$	Explicit, provided
A sequence of input patterns $\mathcal{I}Set$	Explicit, provided
A corresponding set of output patterns $\mathcal{O}Set$	Explicit, provided
SPTTNSimulate requirements of Algorithm-5	Context dependent
SPTTNAdapt requirements of Algorithm-6	Context dependent

Table 10: Input requirements for Algorithm-7

Algorithm 7 $SPTTNTrain(\Xi^{\Xi_{\circlearrowleft}}, \mathcal{I}Set, \mathcal{O}Set), \Xi_c^{\Xi_{\circlearrowleft}} opies^{desir\vec{e}dfire}, \eta, \alpha, \vartheta, t^{start}, t^{inc}, \Upsilon, t^{len})$

$\epsilon \leftarrow \epsilon^{bound} + 1$
while $(\epsilon > \epsilon^{bound})$ **do**
  $\epsilon \leftarrow 0$
  for $(i \in \{1 \dots (|\mathcal{I}Set|)$ **do**
    $t^{start} \leftarrow 0$
    $\mathcal{I} \leftarrow \mathcal{I}Set_i$
    $\mathcal{O} \leftarrow \mathcal{O}Set_i$
    $SPTTNSimulate(\Xi^{\Xi_{\circlearrowleft}}, \mathcal{I})$ [Algorithm-5]
    for $c \in \{1 \dots |copies|\}$ **do**
      $copies[c]_{o^{desir\vec{e}dfire}} \leftarrow \mathcal{T}_{out}^{encode}(\mathcal{O}_c)$
    end for
    for $c \in \{1 \dots |copies|\}$ **do**
      $e \leftarrow e + error(copies[c]_o, copies[c]_{o^{desir\vec{e}dfire}})$
    end for
    $SPTTNAdapt(\Xi^{\Xi_{\circlearrowleft}})$ [Algorithm-6]
  end for
end while

Requirement	Type
A feed forward network with two layers $net \leftarrow \Xi$	Explicit, provided
A sequence of inputs $\mathcal{I}$	Explicit, provided
A temporal encoder $\mathcal{T}_{in}^{encode}$ for I	Explicit, provided
Lower weight bound w^{min}	Explicit, provided
Upper weight bound w^{max}	Explicit, provided
A learning rate η	Explicit, provided
Learning function width β	Explicit, provided
Learning function y-translation b	Explicit, provided
Learning function center c	Explicit, provided

Table 11: Input requirements for Algorithm-7

Algorithm 8 $CNTrain(\Xi, w^{min}, w^{max}, \mathcal{I}, \eta, \beta, b, c)$

for $q \in \{1 \dots |\mathcal{I}|\}$ **do**
  $FFSNSimulate(\Xi, \mathcal{T}_{in}^{encode}(\mathcal{I}_q))$
  $j \leftarrow \Xi^{winner}$
  for $i \in \iota$ **do**
    for $k \in \{1 \dots m\}$ **do**
      $w_{ij}^k \leftarrow w_{ij}^k + L((t_i^a + k) - t_j^a)$
      if $w_{ij}^k > w^{max}$ **then**
        $w_{ij}^k \leftarrow w^{max}$
      else if $w_{ij}^k < w^{min}$ **then**
        $w_{ij}^k \leftarrow w^{min}$
      end if
    end for
  end for
end for

B Design

B.1 Software Requirements Specification

The overall aim is to develop a suite of tools for the purpose of training feed-forward networks of spiking neurons with recurrent connections, aka Recurrent Spiking Networks (RSN), to induce the structures of Moore machines (MM) in their resultant weight configurations. Once trained, the behaviour of an RSN will be examined in order to extract the structure inherent in its resultant weight configuration so that it may be compared to the MM from which the original training set was derived. For full detail, read the main body of the document. It shall be possible to arbitrarily configure the RSN training paradigms RSNTrain (§3.4), and SPTTNTrain (§3.5), and with regard to this, a software requirements specification (SRS) follows, references to the main text are provided in brackets where appropriate.

1.0 FFSN Requirements (§2.2,§3.3)

1.1 Reference Neuron Requirements (§4.3)

1.1.1 It shall be possible to specify the number of reference neurons.

1.1.2 It shall be possible to define the relative firing times of reference neurons.

1.2 Neuron Requirements (§2)

1.2.1 It shall be possible to specify the response function width τ for all neurons.

1.2.2 It shall be possible to define a stereotyped threshold for all neurons.

The following requirements are stereotypical for every neuron to neuron connection:

1.3 Neuron to Neuron Connection Requirements

1.3.1 It shall be possible to specify the number of synapses per connection.

1.3.2 It shall be possible to specify the minimum axonal delay.

Two inhibition models will be available:

1.3.3 Available Inhibition Models

1.3.3.1 It shall be possible to select PSP type on a per neuron basis.

1.3.3.2 It shall be possible to select PSP type on a per synapse basis.

1.3.4 It shall be possible to specify the total fraction of IPSP present.

The following requirements are pertinent to individual synapses:

1.3.5 Per Synapse Requirements

1.3.5.1 It shall be possible to specify an associated synaptic efficacy (weight).

1.3.5.2 It shall be possible to specify an associated axonal delay

1.4 It shall be possible to randomly set network weights from a user defined interval of $\Re$

1.5 It shall be possible to specify the number of neurons in each network layer. (§4.5.3)

1.6 Tools to encode MM inputs over FFSN inputs shall be available. (§4.3)

1.7 Definable MM Input To FFSN Input Encoding Parameters

1.7.1 The number of neurons used to encode each bit.

1.7.2 The value used in the alternating firing time encoding for **low**.

1.7.3 The value used in the alternating firing time encoding for *high*.

1.8 Tools to encode MM outputs over FFSN outputs shall be available.

1.9 Definable MM Output To FFSN Output Encoding Parameters

1.9.1 The number of neurons used to encode each bit.

1.9.2 The value used in the alternating firing time encoding for **low**.

1.9.3 The value used in the alternating firing time encoding for *high*.

1.a Tools to decode FFSN outputs to MM outputs shall be available.

2.0 Definable RSN Recurrency Parameters (§3.4,§4.4)

2.1 The initial imposed state-input firing times.

2.2 The layer from which recurrent connections are taken (recurrent-from layer).

2.3 The amount to delay the firing times of the recurrent-from neurons by.

2.4 Whether or not the delay should be acheived by means of an FFSN or arithmetic.

3.0 Definable Learning Parameters (§2.3,§4.5)

3.1 The learning rate η.

3.2 Whether η should vary dynamically.

3.3 Definable Variable Learning Rate Parameters

3.3.1 The number of neurons to average over.

3.3.2 The value, which if exceeded by the average, indicates oscillation.

3.3.3 The value, which if not exceeded by the average, indicates a plateau.

3.3.4 The amount to decrease the learning rate by if oscillation is detected.

3.3.5 The amount to increase the learning rate by if a plateau is discovered.

4 It shall be possible to to save and load networks to a file or print them to the screen.

5.0 Moore Machine (MM) Requirements (§4.1)

5.1 There must be an unambigious way to define an MM.

5.2 It shall be possible to compare a training set for consistency against a particular MM.

5.3 A mechanism will be provided to randomly generate a test set for an MM.

6.0 RSN Training Requirements (§3.4,§4)

6.1 Training Set Requirements

6.1.1 It must be possible to specify the input and output sequences consistent with the operation of a target MM in an unambigious manner.

6.1.2 It must be possible to specify multiple sub-sequences in a training set.

6.1.3 Implicit and explicit reset symbols (See §4.5.4) shall be defined for separating sub-sequences of an RSN training set.

6.2 Definable Simulation Parameters

6.2.1 Simulation duration.

6.2.2 Time increment value.

6.2.3 Temporal interval between successive inputs (intra-input interval).

6.2.4 The number of training epochs.

6.3 Available Simulation Logging Options

6.3.1 Per epoch squared error.

6.3.2 Per pattern squared error.

6.3.3 Average squared error per symbol per epoch.

6.3.4 Average squared error per symbol per pattern.

6.3.5 The running minimum average squared error per symbol per epoch.

6.3.6 The running minimum squared error per epoch.

6.3.7 The learning rate.

7 An RSN training interface consistent with that described in §3.4.2 shall be provided.

8 An RSN training interface consistent with that described in §3.5 shall be provided.

9.0 Definable Clustering Network (CN) Parameters (§3.6,§5)

9.1 b

9.2 c

9.3 β

9.4 η

9.5 τ

10.0 The first four simulation requirements for the FFSN above shall be met for CN simulation

---

B.2 Risk Analysis

> "If we knew what we were doing, it woudln't be called research, would it?" – Albert Einstein (1879-1955)

It is possible to speculate on whether or not interesting results will be discovered, as similar research has been performed with rate coded networks, but rate and pulse coded networks are fundamentally different, and nobody has tried exactly what will be done here before. However, according to [24], the purpose of the final year project is for the student to demonstrate and apply the knowledge obtained from the rest of the degree, hence, it is not necessary to speculate on whether or not the research will produce anything interesting, only whether there is sufficient scope to demonstrate competence as a computer science professional.

The programming is non-trivial and is large enough that the use of appropriate software engineering practices is a necessity if the project is going to be anywhere near manageable. And it is intellectually at the right level for a third year undergraduate project.

A possible risk is that the code or the documentation will not be completed in time. This will be tackled through effective time management, and the culturing of appropriate motivation. The student possesses a genuine interest in science, and is particularly interested in the area, thus motivation should not be a problem, however special effort shall be made to maintain motivation by keeping up with the relevant research material, and seeking out discussion with similar minds. Relaxation techniques shall be practiced to maintain productivity.

During the summer of 2003, the student experienced some RSI problems, has subsequently had MRI scans in an attempt to determine the cause of the problems, and is awaiting nerve conduction velocity tests. Thus, a very prominent risk is that the student's RSI could flare up and hinder the production of code or documentation, and possibly detriment motivation due to consequent feelings of not being able to control the situation. Thus care must be taken to manage the amount of typing done, and special care taken to maintain a correct operating posture. The documentation phase should be started as early as possible so that it may be sufficiently spaced out so that median nerve inflammation does not occur due to excessive typing. The appropriate authorities shall be informed of possible problems, in advance of their possible occurrence, so that in such a case mitigations can be considered.

Another risk is that of hardware failures and accidents that may damage source code or documentation. This must be catered for by taking regular backups and avoiding the typing of command sequences that, if mistyped, could result in catastrophe, for example, using example "rm *∼" to delete scratch files is dangerous because if the final '∼' were missed out, all the files in the current directory will be deleted if owned by the user and noclobber is unset.

The risks discussed in this section along with the suggested solutions are presented in tabular form below:

Risk	Proposed action
Inability to produce scientifically interesting results.	Irrelevant due to capacity to demonstrate computer science prowess intact.
Lack of motivation resulting in non-attainment of goals.	Pro-active research, relaxation techniques, association with similarly inclined individuals.
RSI problems may result in reduced capacity to perform computer related tasks.	Manage computer use effectively, follow professional suggestions. Reserve ample time for completion of tasks involving heavy typing.
Catastrophic hardware failure, and accidental corruption of active research material.	Regular backups, avoiding easily mistyped dangerous command sequences.

B.3 Use cases

B.3.1 Use case introduction

The design goal is develop a suite of tools for training feed forward networks of spiking neurons (Feed forward Spiking Neuron networks FSN) with recurrent connections (Recurrent Spiking Neuron networks RSN) to induce the structures of Moore machines (MM) in their weights, and tools to extract the induced structures for comparison with target MM, a Clustering Network (CN) is used in the process. That is, a MM is first *induced* then *extracted*. A use case diagram is shown in Figure-17.

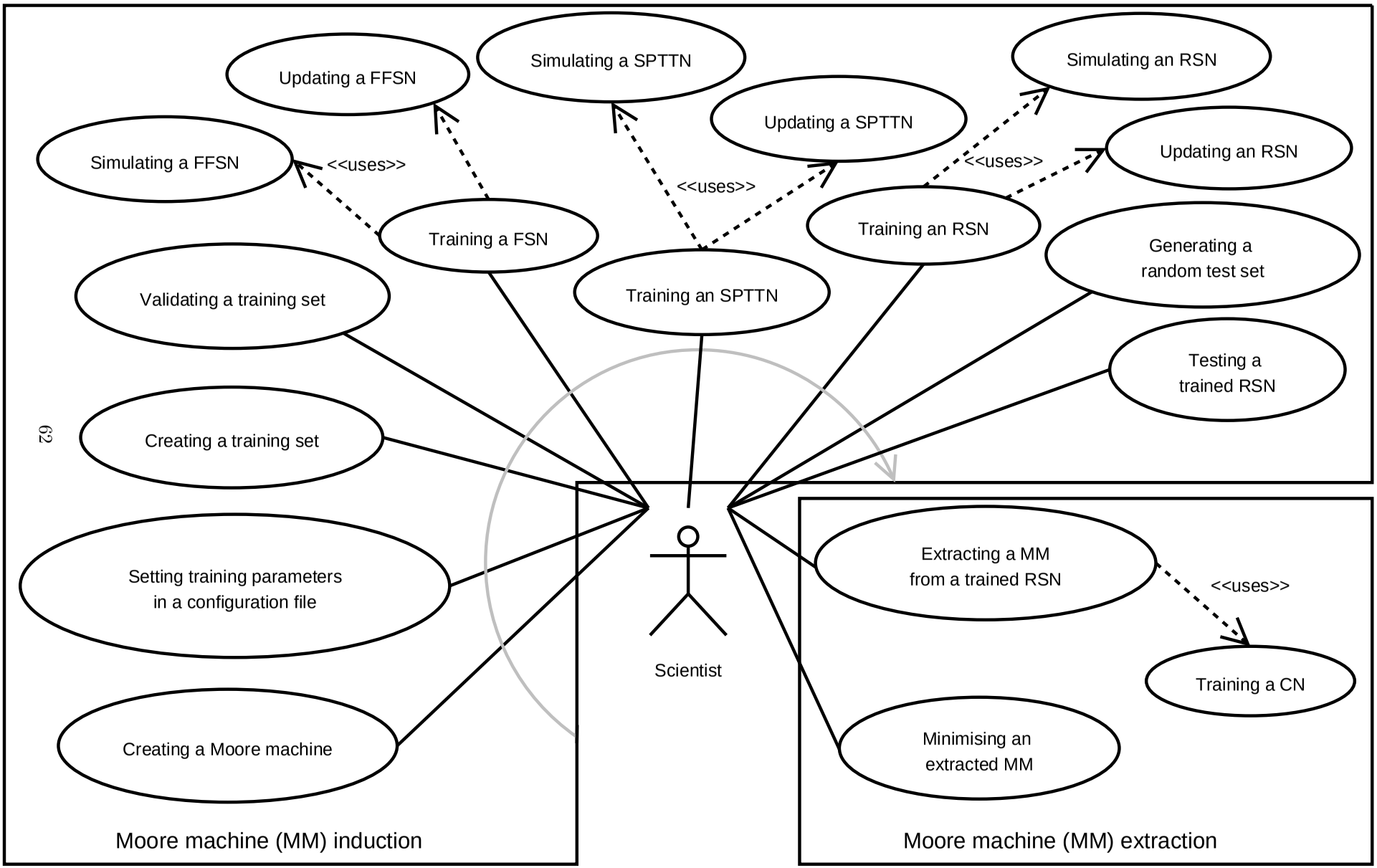


Figure 17: Use Case Diagram

B.3.2 Actors

The only actor involved is a scientist. Initially the only scientist that will be using these tools is the author of this document.

B.3.3 Pre-conditions

A trained RSN and target MM is required to extract an MM from the RSN.

B.3.4 Post-conditions

After MM induction a target MM should have been chosen and a trained RSN obtained from training on a training set derived from the MM. After MM extraction a minimal MM should have been extracted from the trained RSN.

B.3.5 Primary Path

The primary path pertains inducing and extracting MM from trained RSN. Events occur in a clockwise direction with respect to what is drawn in Figure-17, starting at the bottom left and ending at the bottom right, as indicated by the circular grey arrow. Thus in brief, a target MM is created, training configuration parameters are setup, a training set is created, then validated against the target MM, at this point three training possibilities occur, only the latter two are are applicable to MM induction, thus either an SPTTN (SpikeProp-Through-Time Network) is simulated and updated multiple times, or an RSN is. An SPTTN will contain an RSN, thus the result is a trained RSN. A random test set is generated for the trained RSN, and if the network performs well then it indicates that the target MM has been induced, hence extraction commences and this involves training a ClusteringNetwork (CN), the end result is an extracted Moore machine MM, which can be minimised.

B.3.6 Alternative Path

The user performs any of the use cases arbitrarily, for example, training an FFSN.

B.4 Class diagrams

The suite of tools that entail induction and extraction of Moore machines, consists of approximately 12 different programs written in c; a non-object-oriented language. This makes it quite hard to use official UML notation everywhere. Thus a UML themed software diagrammatic style augmented with verbal explanation is used instead, foreign concepts will be explained where necessary.

B.4.1 Network Oriented Classes

B.4.1.1 Network Class Diagrams

This section describes the classes and binaries involved in the simulation of the various networks used in the induction and extraction processes.

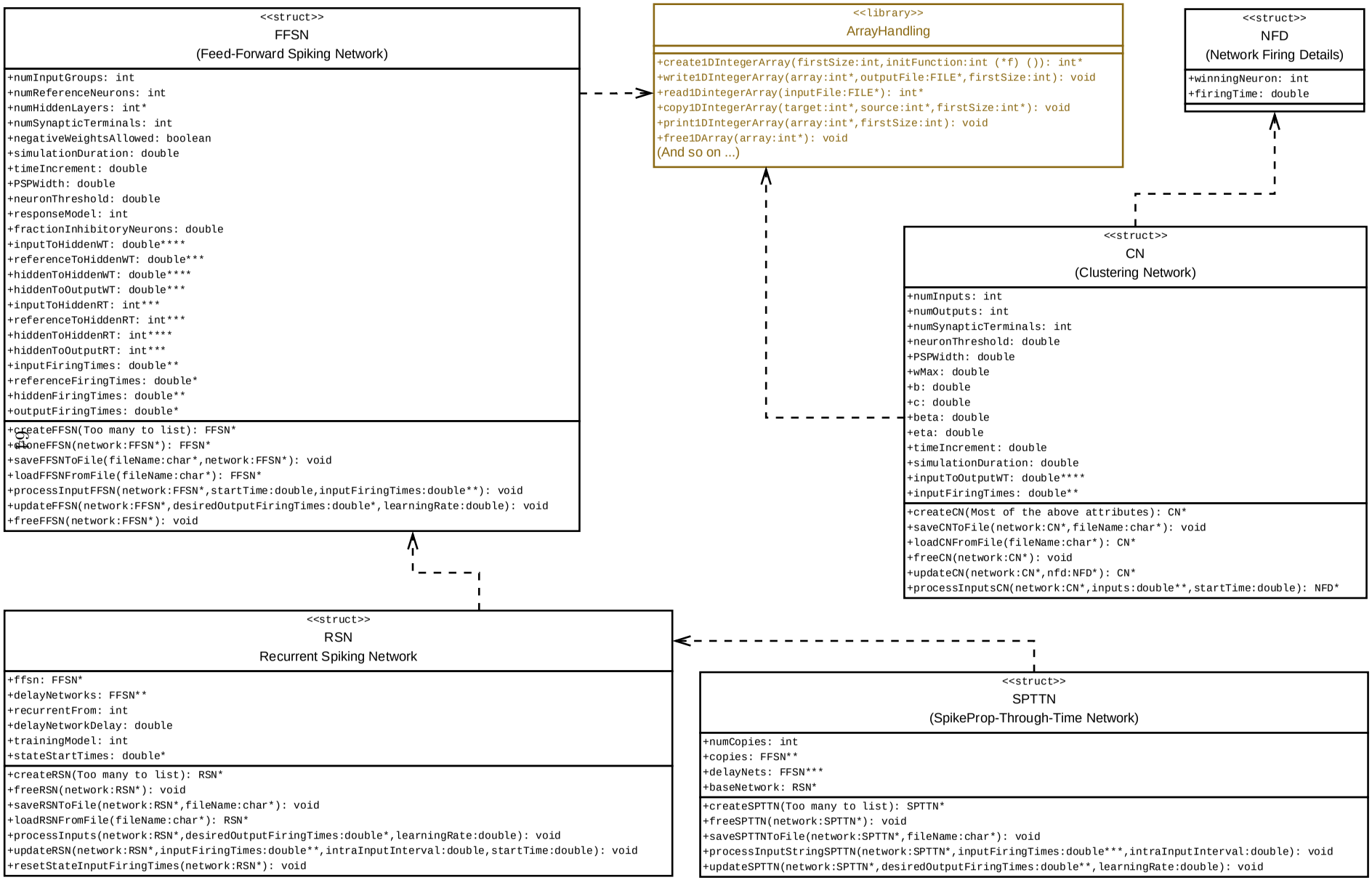


Figure 18: Network Related Class Diagram

Figure-18 shows a UML class dependency diagram, dashed lines represent dependencies, and it uses the stereotypes `<<struct>>`, for c structure and associated functions, `<<exe>>` for executable programs, `<<exe,test>>` for executable test programs, and `<<library>>` for library-like code components. Dependencies are presumed to be transitive. It contains the following classes:

1. **ArrayHandling** is a library-like class that provides all array handling routines, such as creation, saving to file, loading from file, printing, copying, and destroying. It provides functions to create arrays of type int and double.
2. **FFSN** (Feed-Forward Spiking Network) (For theory see §2.2) provides the basic feed-forward network simulation (For theory see §3.3) and update mechanisms which are the core of the entire system. It depends on ArrayHandling for its multi-dimensional array creation and destruction routines. Its attributes are simulation and updating parameters. Its most important methods are the simulation and update routines.
3. **RSN** (Recurrent Spiking Network) (For theory see §3.4) wraps a FFSN to provide recurrent network capabilities, it thus depends on FFSN. RSN are trained to induce Moore machines.
4. **SPTTN** (SpikeProp-Through-Time-Network) (For theory see §3.5), provides the functionality to train an RSN by expanding it through time. It is thus dependent on RSN, and consequently by transitivity FFSN.
5. **NFD** (Neuron Firing Details) is a data structure for storing the firing time of a winning neuron from a simulation of a CN (See next).
6. **CN** (Clustering Network) is an unsupervised training mechanism, implemented using spiking neuron networks (For theory see §3.6), after its simulation it returns an instance of NFD containing the firing time and index of the neuron which fired first in the simulation, so that this may be passed to its update function.

B.4.1.2 Putting the classes in context

Figure-19 puts the classes described above in context by linking them to the executables involved in Moore machine induction and extraction that they are used to created. The purpose of each executable is then explained in brief.

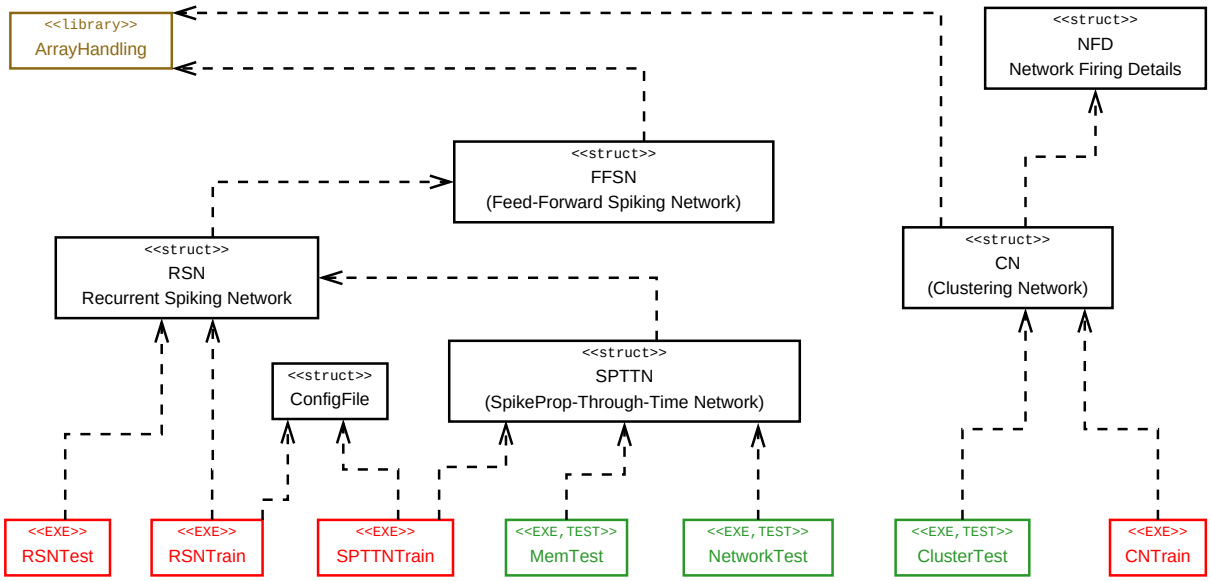


Figure 19: Dependencies between the network oriented classes described in

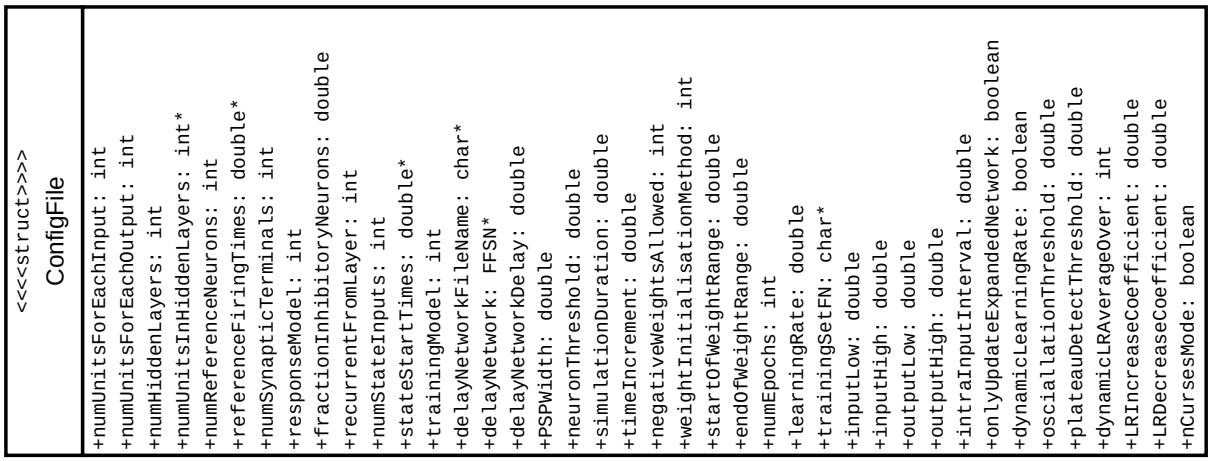


Figure 20: *ConfigFile*:Configuration file for network training

1. *SPTTNTrain* Provides an interface for training RSN in the SPTTN fashion described in §3.5. Uses ConfigFile (See below)
2. *RSNTrain* Provides an interface for training RSN in the standard manner described in §3.4.2. Uses ConfigFile (See below)
3. *RSNTest* Tests a trained RSN over a test set (MMIOP) for a particular MM (Moore Machine).
4. *CNTrain* Interface to the unsupervised training capabilities of CN; used to extract MM (§5).
5. *DelayNetworkTrain* An RSN can use a network instead of an arithmetic constant to implement its delay function α (See section §3.4.2). This program provides the ability to train an FFSN to implement an arbitrary length delay.

The remaining three programs are used for testing, refer to §B.6 for further information.

B.4.1.3 ConfigFile

The structure ConfigFile is used to store training parameters, it is shown in Figure-20, and used by RSNTrain, and SPTTNTrain.

B.4.2 Extraction Oriented Classes

B.4.2.1 Extraction Oriented Class Diagram

This section describes the classes and binaries involved in the extraction of Moore machines (MM) from trained RSN.

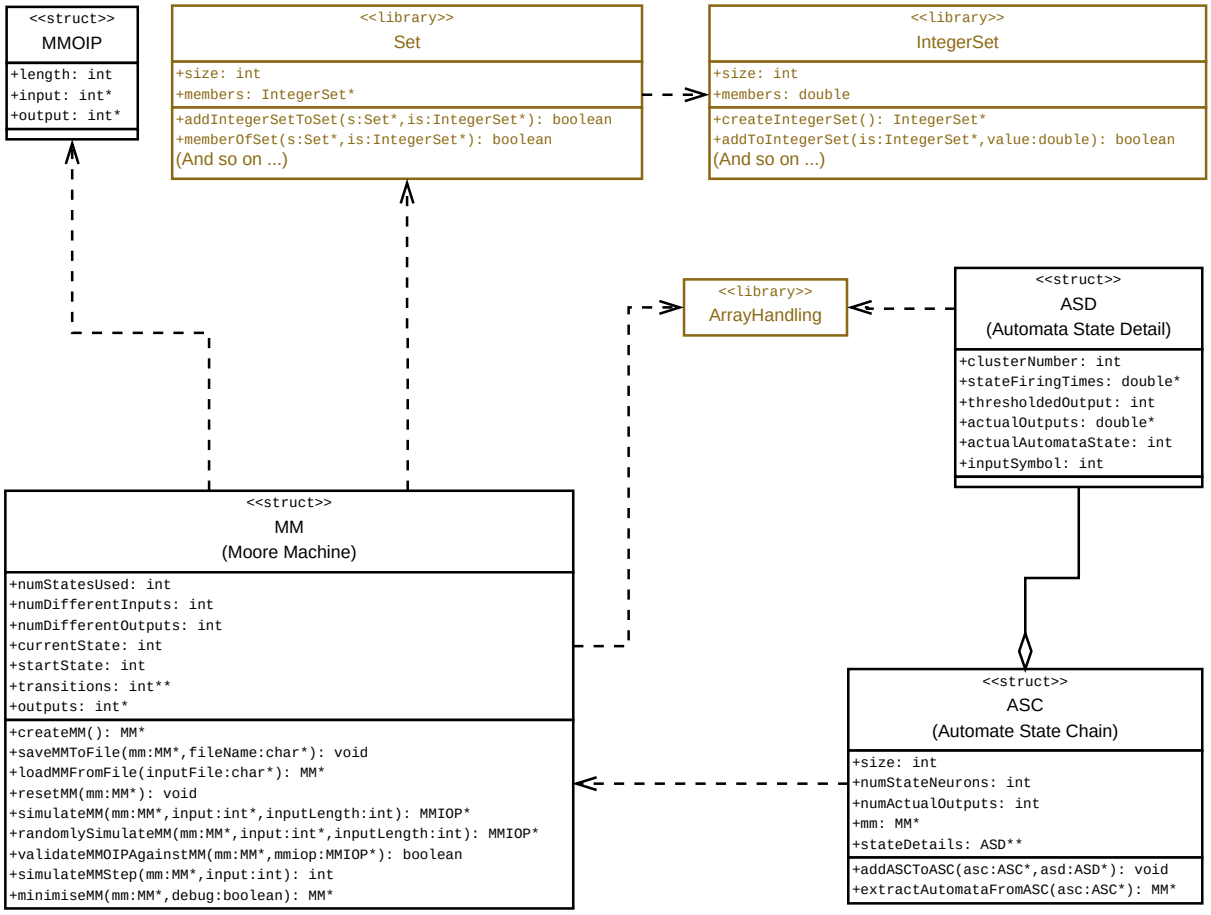


Figure 21: Network Related Class Diagram

Figure-21 shows a UML class dependency diagram, dashed lines represent dependencies, and it uses the stereotypes `<<struct>>`, for c structure and associated functions, `<<exe>>` for executable programs, `<<exe,test>>` for executable test programs, and `<<library>>` for library-like code components. Dependencies are presumed to be transitive. It contains the following classes:

1. **MMIOP** (Moore Machine Input Output Pair) This class represents a training set which can be used to train a RSN, a FFSN, or an SPTTN. It consists of and integer array of inputs, an integer array of outputs, and a training sequence length.
2. **IntegerSet** This library class provides an integer set functionality. It includes support for vectors as well as genuine sets.
3. textbfSet Dependent on IntegerSet, this class provides sets of IntegerSets.
4. **MM** (Moore Machine) This class encapsulates the Moore machine (For theory see §4.1) which is the target of the induction and extraction regimes. It depends on Set and MMIOP.
5. **ArrayHandling** This class was discussed in the last section.
6. **ASD** (Automata State Detail) During the extraction of an MM (See §5), details about each state are required to be stored, this class provides that capability.
7. textbfASC Automata State Chain Used in the extraction process, ASC provides the ability to store multiple ASD and then perform operations on this chain such as extracting an MM. In addition to those methods shown, ASD and ASC have basic loading and saving capabilities.

B.4.2.2 Putting the classes in context

Figure-22 puts the classes described above in context by linking them to the executables involved in Moore machine induction and extraction that they are used to created. The purpose of each executable is then explained in brief.

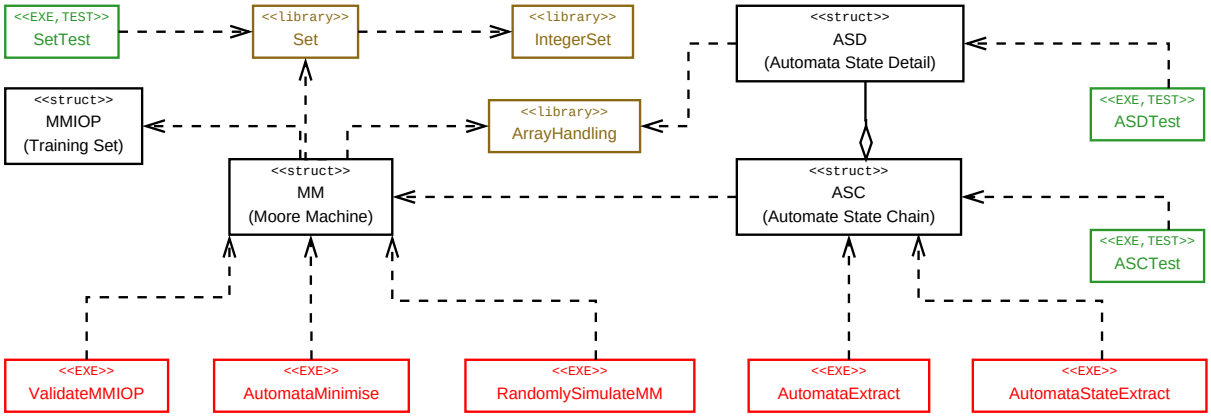


Figure 22: Dependencies between the network oriented classes described in

1. **ValidateMMIOP** provides the capability to validate an MMIOP (training set) against the MM with which it is supposed to correlate.
2. **AutomataMinimise** provides the capability to minimise an extracted MM.
3. **RandomlySimulateMM** allows the creation of arbitrarily sized random MMIOP to be generated from a specific MM, using a random walk over the MM.
4. **AutomataStateExtract** extracts the states and details about the transitions of the structure that has been induced in an RSN.
5. **AutomataExtract** After the states and transition data of an MM have been abstracted by AutomataStateExtract, this program is used to construct a MM from the file produced.

B.5 ControlFlow

Figure-23 shows how the executable programs described in the previous section are used to actualise the primary path described in §B.3.5, i.e training RSN to induce the structures of target MM in their weights then subsequently extracting this induced information to compare with the target MM.

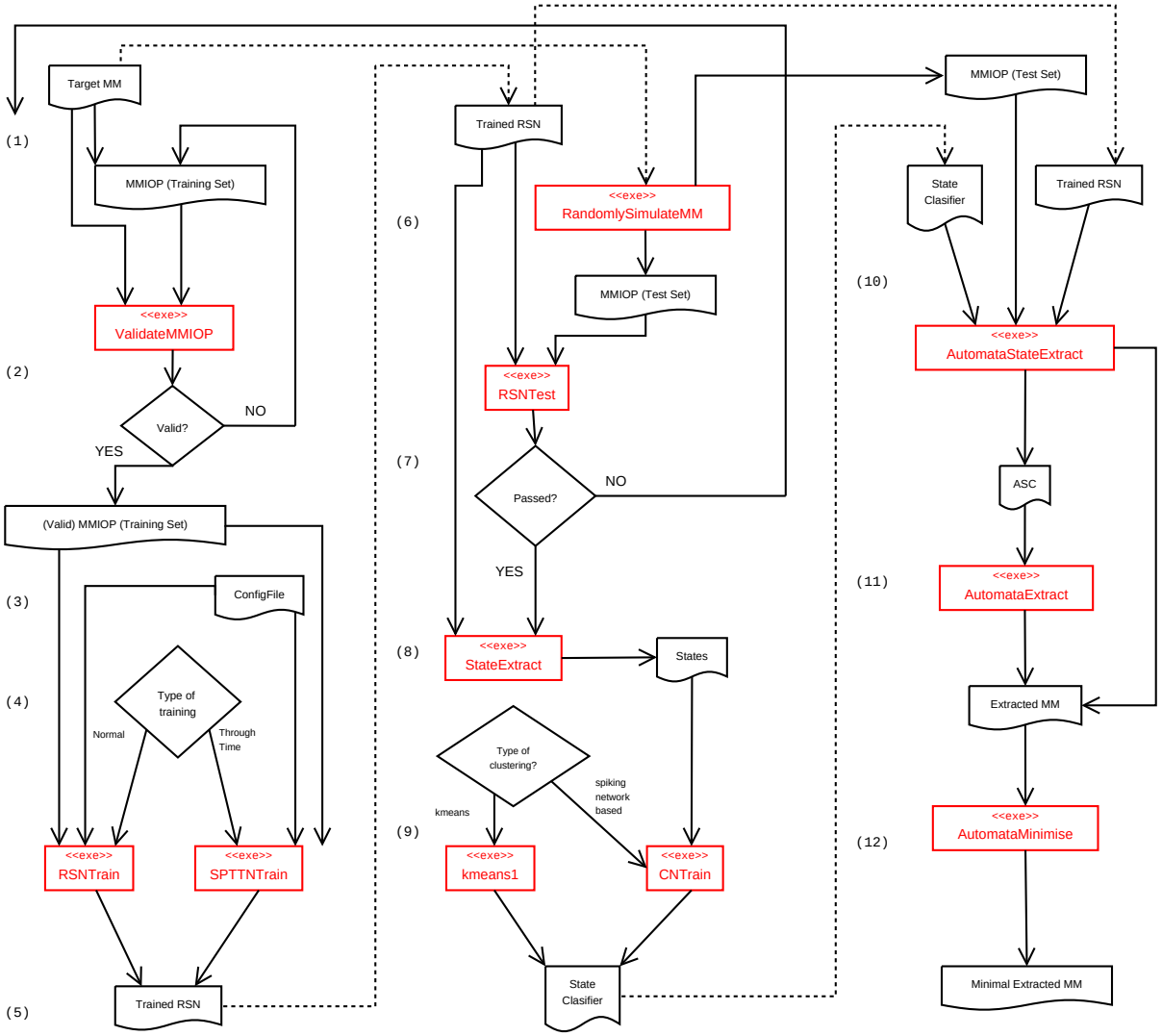


Figure 23: Sequential flow

1. The initial step is to choose a target Moore machine MM (§4.1) and develop a training set for it (§4.5.4).

2. Once a training set has been obtained its output should be compared against the operation of the target MM over its input to check for consistency. If it is not valid then the training set should be revised until it is valid. ValidateMMIOP will print out the index of the first invalid input and output symbols to help this process.

3. Given a valid training set, it is necessary to setup the training configuration. This is done by creating a configuration file which contains the fields of the ConfigFile class, and specifies things like learning rate, network architecture, and so on.

4. Given a valid training set and configuration file, it is time to decide what type of training should be performed. A choice must be made between standard recurrent training (§3.4.2) and SpikeProp-Through time training (§3.5).

5. Once a choice has been made, the network is trained on the training set set to give a trained RSN.

6. There is no point attempting to extract a MM from the trained network if nothing has been induced, thus a large random test set, consistent with the target MM is created with RandomlySimulateMM and then the network is tested on this test set. Note that the test set could also be created manually.

7. The test results are examined, and if the test error is sufficiently low, such that it is likely that there is an interesting induced structure, extraction can proceed at the next step. If the test error is high then either a bad training run occurred or the training set was inappropriate, or some training parameter choices were inappropriate, in this case the user should go back to the start, and keep retraining, and/or adjust parameters, and/or alter the training set until the desired result is achieved.

8. Given that a sufficiently interesting trained RSN has been obtained, the extraction procedure can commence (§5). The first step is to extract all the state firing times from the trained RSN to give a file full of states "States".

9. The state file must be clustered to obtain a classifier for arbitrary states. There are two types of clustering currently available, either standard K-means can be used, or the preferable spiking network based clustering.

10. Given a state classifier, the state and transition information of the structure contained in the trained RSN can be extracted. Another test set is required for this, however, what is not shown in the diagram is that can either be explicitly automatically created by randomly walking over the target MM, or it can be created implicitly via the interactive mode of AutomataStateExtract, or it can be explicitly manually created. There is a link from AutomataStateExtract directly to Extracted MM, because in interactive mode the user *can* extract an MM manually, but the user is not *required* to. If the input is not provided interactively, then the test set used, whatever input vector is used, the input is fed into AutomataStateExtract to obtain an ASC (Automata State Chain) which contains details about the states and transitions visited by the the induced structure over the test set.

11. Given an ASC, containing a list of states and transitions, AutomataExtract is used to extract an MM from this list.

12. The extracted MM might not be minimal, in which case it can be fed through AutomataMinimise to make it minimal.

B.6 Testing

B.6.1 General Strategy

The general testing strategy was to use standalone tests where it was necessary t ensure that something was working. In addition several tests were scripted so that the script could be run, and if any test failed the script would indicate this in its output, the main reason for this was to check that things were still working after the system had been incrementally modified to make it easy to track bugs so that they did not get lost amongst other bugs through time, i.e their presence can be attributed to the modifications between last successful testing and the current testing. The scripted tests were ASDTest, ASCTest, AutomataMinimiseTest, and NetworkTest (see below).

B.6.2 Miscellaneous Testing

The following programs, whose relations to the rest of the system are shown in Figure-19, are used to test the network components of the system.

1. *MemTest* Tests that memory allocated when a FFSN, RSN, or SPTTN is created, is correctly freed.

2. *NetworkTest* Tests the checks that when a FFSN, RSN, or SPTTN is saved, it can be loaded again and vice-versa, this test is also automated.

3. *ClusterTest* Tests the clustering abilities of CN on an artificially constructed clustering task.

The following programs, whose relations to the rest of the system are shown in Figure-22, are used to test the network components of the system.

1. **SetTest** Tests the capability of Set and IntegerSet.

2. **ASDTest** Tests the creation, loading, and saving of ASD, this test is also automated.

3. **ASCTest** Tests the creation, loading, and saving of ASC, this test is also automated.

The following programs are not shown in class context in any figures, but are also used for testing:

1. **XORExperiment** Tests that an FFSN can actually operate correctly in a feed-forward manner.

2. **AutomataMinimiseTest** This is a script which tests that Minimisation is working, it compares hand computed minimisations of MM against computer minimisations of the same MM for equality.

B.6.3 Clustering Network Testing

A data set containing 4 clusters was generated by Dr Peter Tiňo. Each cluster consisted of 1000 elements of $\Re^2$, the clusters are shown visually in Figure-24. The first 100 data elements from each cluster were encoded in sequential runs over the input neurons of a clustering network with 4 output neurons using the distributive encoding method of [1], the network was then simulated and updated as in §3.6 with $b = -0.2$, $c = -1.5$, $\beta = 1.67$, $\eta = 0.01$, $\vartheta = 1$, and $\tau = 3$.

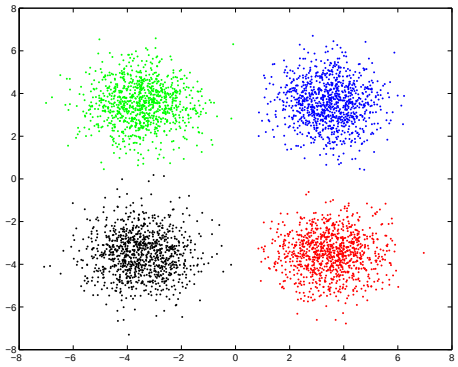


Figure 24: Four clusters for testing the clustering network implementation

The resultant network was used to classify all elements of the original data set. The output neuron which fired most frequently for inputs from each cluster, was considered to be the identifier for that cluster. If there was not a one to one mapping between clusters and output neurons, as implied by the frequency of firing, then the procedure was restarted with a fresh clustering network. Once the one to one mapping had been obtained between each cluster and output neuron. The amount of elements of the data set classified into the correct cluster was recorded. The procedure was repeated 100 times and the average classification error per cluster recorded. It was negligible.

B.7 Project Management

Several things were done to make managing the project easier. Firstly the whole project was implemented under CVS, and secondly incremental testing was performed to prevent breakage. In terms of temporal management, no temporal schedule was constructed at the beginning of the project; as it was research oriented, concrete temporal plans would not be applicable as the path of the research was unpredictable, and it happened that research direction changed several times owing to observed results (See below). However, during the writeup stage, a schedule was constructed approximately half way though to help meet deadlines, although one component took twice as long to write up than was projected, probably because the author was inexperienced in temporal planning. The documentation was completed on time and to a high standard, and it is thought that the schedule aided the process significantly.

In the beginning it was thought that a naive monolithic program might suit the needs for the research, since it was thought that results would not be that hard to achieve, and coupled with a lack of real understanding of how to go about designing the system due to having no previous experience in the domain, parts of the software were first implemented in such a manner. The initial models failed to produce anything considered as an interesting result, so several different RSN training models were tried, and there ended up being several different versions of the code around as a consequence. It was very hard to manage the several versions concurrently. A lack of results considered interesting even after trying several models, accompanied by a lack of c programming experience resulted in a bug paranoia developing. Thus much time was spent proof reading the code repeatedly looking for bugs, a few were found but they were non-critical, and fixing them did not improve the success rate or otherwise obtain any new results.

It was only after substantial experience with the domain that it became clear what the requirements were and how the system should be implemented to obtain an extensible and more useful system, thus much of it was redesigned again and reimplemented. Accompanied with this was a new intuition about how to design the experiments, and a new intuition about the previously considered uninteresting results of the former executed experiments, it was at about this time that the first results were obtained, or more accurately, were obtained and realised as being results. Previously results had been obtained, but because of the bug paranoia and complexity of the systems in use, it had become so confusing that it was not clear which system had obtained the results, or if that system had a bug in it when the results were obtained. It was only after the design was formalised, and the system made more manageable and easy to maintain, that repeated proof reading and testing diminished the bug paranoia, and then the previously considered insignificant results, became significant, because it became evident that it was unlikely that there was a problem with the systems in use, and more likely that actually the task was very hard.

I am not convinced that blindly following software engineering practices from the start would have been a better idea. Without really understanding the problems associated with **not** using a formal design process, I believe that I would have made same or similar mistakes anyway, primarily because I would not have really understood *from experience* why I was doing what I was doing, because I had never implemented a system of such magnitude before, and had never felt the effects of design mismanagement first hand to such a great extent. I have learned a valuable lesson from this which I am sure will greatly help me the next time I implement a large software project. I am now fully convinced of the worth of software engineering and formal design because I have directly experienced the results of not applying them, and then saw things get better and better as soon as they were applied. I can also see how the same approaches would be worthwhile applied to documentation too. Thus this has been a very beneficial learning experience, and it has had even more of an impact since I have learned the hard way.

B.8 Appraisal

All the requirements of §B.1 were met in the final design. The software was reasonably efficient as it was programmed in C, so this was a slight advantage over C++ and a large advantage over Java, but by programming in C, although object-oriented concepts were used in the design, they had to be implemented in a non-object-oriented language which limited their effectiveness. However, the overall

software is robust, fully functional, and extensible.

The final report has turned out extremely large. It is not currently obvious to the author how its size could be reduced without sacrificing clarity, omitting important points, or detrimenting readability. This is attributed to the inexperience of the by author in such matters.

C Experimental Parameter Listing

Parameter	Value
Number of neurons each input is encoded over	5
Number of neurons each output is encoded over	1
Number of hidden layers	3
Number of units in each hidden layer	5 5 5
Number of reference neurons	2
Reference relative firing times	2.000000 2.000000
Number of synaptic terminals per connection	16
Response model	0
Fraction of connections that are inhibitory	0.200000
Hidden layer that recurrent-from comes from	1
ϖ^{start}	2.000000 6.000000 3.000000 6.000000 2.000000
Training model	0
Δ^t	13.000000
Response function decay constant τ	3.000000
Neuron threshold ϑ	50.000000
Simulation duration	30.000000
Simulation time increment	0.100000
Allow negative weights?	0
Weight initialisation method	0
Start of weight range	1.000000
End of weight range	1.000000
Number of epochs	1000000
Initial learning rate	1.000000
Low input encode value	2.000000
High input encode value	6.000000
Low output encode value	20.000000
High output encode value	26.000000
Intra-input interval Υ	20.000000
Dynamic learning rate	1
Oscillation threshold	0.500000
Plateau threshold	0.050000
Number of epochs to average over for dynamic learning	7
Learning rate increase coefficient	1.010000
Learning rate decrease coefficient	0.990000

Table 12: Training parameters for the TwoState Moore machine induction experiment discussed in §6.1 and the ThreeState induction experiment discussed in §6.2

D Acknowledgments

First off I would like to thank Dr. John Bullinaria for encouraging me to apply for a Nuffield Science Foundation research bursary in the second year of my degree, and for suggesting Peter Tiňo as a supervisor.

Secondly I would like to thank Dr. Peter Tiňo for being an excellent research collaborate, friend, and mentor. Without his continued encouragement, affinity for hard work, and unfaltering determination, when times were tough, I might never have made it.

I would also like to thank Blackwell Publishing, and Oxford University press for granting me permission to use, respectively, the right hand side of Figure-1, and Figure-2. I especially commend them for their rapid response to my permission request, which was submitted at respectively two days and one day before the dissertation deadline.

I would like to thank the authors of the many free software tools and the authors of the free operating system FreeBSD, that I have utilised throughout this research. There is no doubt that without these indispensable tools, my path would have been an order of magnitude more arduous.

Finally I would like to thank everyone else who has supported, tolerated, or otherwise assisted me in any way over the past six months. Although you may never know it, your help was nevertheless invaluable.

E Running The Code

This appendix is a requirement of the dissertation submission. It should not be considered as part of the dissertation proper.

The source code is located in: "~ug55axm/finalyearproject/src".

There is no single executable program to demonstrate, but instead a whole suite. It is not obvious then how to demonstrate that all the programs work, because this would take up several pages, and the guidelines state this appendix cannot be more than one page long. So that something may be demonstrated, if the following sequence of commands can be followed then the simple feed-forward learning of a temporal XOR problem with spiking networks obtains:

1. ssh into bro (This was the test machine so it definitely works here): `ssh bro`
2. cd to the authors home directory: `cd ~ug55axm/finalyearproject`
3. delete the binaries in `bin`, to exclude the possibility they have been specially manufactured: `rm bin/*`
4. cd to the source directory: `cd src`
5. make the xor test program: `make`
6. cd to the xor test directory: `cd ../xor`
7. run the test script: `./Run`, press the keyboard character '`c`', and watch.
8. The actual firing times, shown in the upper left corner of the terminal, should converge to the displayed desired firing times in about 500 epochs. If it is not obvious that the network is converging or has converged to the solution after 500 epochs, press `q` and go back to step 7 to try different initial weights. Otherwise press `q` to quit.

If you try and compile anything else, be warned, **make** will probably complain about a missing function called **srandomdev**, this is a special random number seeding system which uses an entropy device to generate very good random numbers. It is not available on the computer **bro**. The code was designed to run on FreeBSD. If you have any questions or would like to see more demonstrations please do not hesitate to contact the author: **ug55axm@cs.bham.ac.uk**. Thanks.

References

[1] Sander Marcel Bohte. *Spiking Neural Networks*. PhD thesis, Centre for Mathematics and Computer Science (CWI), Amsterdam, 2003.

[2] Patricia S. Churchland and Terrence J. Sejnowski. *The Computational Brain*. MIT Press, 1993.

[3] D. Chen H.H. Chen G.Z. Sun C.L. Giles, C.B. Miller and Y.C. Lee. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3):393–405, 1992.

[4] A. Cleeremans, D. Servan-Schreiber, and J.L. McClelland. Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381, 1989.

[5] H. H Dale. Phaarmacology and nerve endings. *Processdings of the Royal Society of Medicine*, 28:319–332, 1935.

[6] S. Das and M.C. Mozer. A unified gradient–descent/clustering architecture for finite state machine induction. In J.D. Cowen, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 19–26. Morgan Kaufmann, 1994.

[7] Ariel Y. Deutch and Andrew J. Bean. Colocalization in dopamine neurons, 2000.

[8] John Carew Eccles. *The Physiology of Nerve Cells*. Oxford University Press, 1957.

[9] John Carew Eccles. *Physiology Of Synapses*. Springer-Verlag, 1964.

[10] Ken-Ichi Funahashi. On the approximate realization of continous mappings by neural networks. *Neural Networks*, 2(3):183–192, 1989.

[11] Gray E. G. Electron microscopy of presynaptic organelles of the spinal chord. *Journal Of Anatomy*, Volume 97:101–106, 1963.

[12] Y. Wang H. Markram and M. Tsodyks. Differential signaling via the same axon of neocortical pyramidal neurons. *Neurobiology*, 95:5323–5328, April 1998.

[13] L.H. Hamlyn. The fine structure of the mossy fibre endings in the hippocampus of the rabbit. *Journal Of Anatomy*, 96:112–120, 1963.

[14] W. Maass. On the computational power of winner-take-all. *Neural Computation*, 12(11):2519–2536, 2000.

[15] W. Maass and E. D. Sontag. Neural systems as nonlinear filters. *Neural Computation*, 12(8):1743–1772, 2000.

[16] Wolfgang Maass. Lower bounds for the computational power of networks of spiking neurons. *Electronic Colloquium on Computational Complexity*, 1(19), 1994.

[17] Wolfgang Maass. Fast sigmoidal networks via spiking neurons. *Neural Computation*, 9(2):279–304, 1997.

[18] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997.

[19] Wolfgang Maass and Christopher M. Bishop, editors. *Pulsed Neural Networks*. MIT Press, 2001.

[20] Wolfgang Maass and Thomas Natschläger. Networks of spiking neurons can emulate arbitrary hopfield nets in temporal coding. *Network: Computation in Neural Systems*, 8(4):355–372, 1997.

[21] W. McCulloch and W. Pitts. A logical calculus of the ideas imminent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[22] M. Minksy and S. Papert. *Perceptrons (Expanded edition)*. MIT Press, 1988.

[23] Simon Christian Moore. Back propagation in spiking neural networks. Master's thesis, The University of Bath, 2002.

[24] School of Computer Science. Guidance notes for undergraduate final year projects and msc computer science projects, 2003/04.

[25] Han La Poutré Sander M. Bohte, Joost N. Kok. Error-backpropagation in temporally encoded networks of spiking neurons. *ELSEVIER Neurocomputing*, 48(1-4):17–37, October 2002.

[26] Wolfgang Maass Thomas Natschläger. Spatial and temporal pattern analysis via spiking neurons. *Network: Computation in Neural Systems*, 9(3):319–332, August 1998.

[27] Wolfgang Maass Thomas Natschläger. Spiking neurons and the induction of finite state machines. *Theoretical Computer Science: Special Issue on Natural Computing*, 287(1):251–265, September 2002.

[28] P. Tiňo and J. Sajda. Learning and extracting initial mealy machines with a modular neural network model. *Neural Computation*, 7(4):822–844, 1995.

[29] R.L. Watrous and G.M. Kuhn. Induction of finite-state languages using second-order recurrent networks. *Neural Computation*, 4(3):406–414, 1992.

[30] V. P Whittaker and E. G. Gray. The synapse: Biology and morphology. *Brit med. Bull*, 8:223–328, 1962.

[31] Z. Zeng, R.M. Goodman, and P. Smyth. Learning finite state machines with self–clustering recurrent networks. *Neural Computation*, 5(6):976–990, 1993.