ENGINEERING A SEQUENCE MACHINE THROUGH SPIKING NEURONS EMPLOYING RANK-ORDER CODES

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2007

By
Joy Bose
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

Sequence memories play an important role in biological systems. For example, the mammalian brain continuously processes, learns and predicts spatio-temporal sequences of sensory inputs. The work described in this dissertation demonstrates how a sequence memory may be built from biologically plausible spiking neural components. The memory is incorporated in a sequence machine, an automaton that can perform on-line learning and prediction of sequences of symbols.

The sequence machine comprises an associative memory which is a variant of Pentti Kanerva's Sparse Distributed Memory, together with a separate memory for storing the sequence context or history. The associative memory has at its core a scalable correlation matrix memory employing a localised learning rule which can be implemented with spiking neurons.

The symbols constituting a sequence are encoded as rank-ordered N-of-M codes, each code being implemented as a burst of spikes emitted by a layer of neurons. When appropriate neural structures are used the spike bursts maintain coherence and stability as they pass through successive neural layers. The system is modelled using a representation of order that abstracts time, and the abstracted system is shown to perform equivalently to a low-level spiking neural system. The spiking neural implementation of the sequence memory model highlights issues that arise when engineering high-level systems with asynchronous spiking neurons as building blocks.

Finally, the sequence learning framework is used to simulate different sequence machine models. The new model proposed here is tested under varied parameters to characterise its performance in terms of the accuracy of its sequence predictions.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

First of all I would like to thank my supervisors Steve and Jon, who taught me most of what I know about how research is done and complemented each other perfectly in my supervision. Specifically I would like to thank Steve for the knowledge of his engineering background, his inspirational work ethic, for being so methodical in every aspect of supervision despite having so many engagements, and kindling my interest in the field. I would like to thank Jon for sharing the breadth of knowledge from his physics and neural networks background and training me to think precisely as a scientist and also teaching me valuable skills regarding doing repeatable experiments and academic writing (although I cannot claim to have followed all or even most of them!). I thank both of them for many delightful discussions and useful insights and for giving up their valuable time and for giving me an opportunity to explore this research field.

I would like to thank Mike Cumpstey, a research fellow in the APT group, for building the Spiking Neuron simulator and for giving up many hours for my bothering him to make changes or correcting bugs in the simulator, for showing me some ways to think about simulating neurons that I wouldn't normally have thought about and for his modular coding style that was invaluable for me to learn.

I would like to thank all my colleagues in the APT group (and especially in the neural networks subgroup) for providing a good academic environment and support throughout, for honing my skills of giving presentations by giving useful feedback, Viv for so many times persuading me to present my research, Andrew in the first two years and Charlie since, for maintaining all the hardware and software I needed.

For proofreading and correcting this dissertation, I would like to thank again Steve, Jon and Mike.

I would like to thank the Overseas Research Studentship Scheme, funded

by the British Government, for partly financing my study, and the University of Manchester School of Computer Science for financing the rest and also for all their facilities and support, especially the library facilities for the latest computational neuroscience related books and e-journals.

Finally, I would like to thank my parents Sivaprasad and Indrani for being so supportive throughout my study and helping me in every way they could without caring for their personal hardship.

Chapter 1

Introduction

Understanding the functioning of the brain and how it encodes and transmits information has been a subject of great interest. The human brain can perform functions such as sensory-motor coordination, visual recognition of moving objects, auditory frequency discrimination, etc with great speed and accuracy, which seem trivial for humans but difficult to be modelled on computers. The brain is built of component cells called neurons. Information is transmitted through the neurons in the brain largely in the form of action potentials or spikes [2, 44], which are generated by an electrochemical process and regenerated from neuron to neuron. The spikes are similar in shape and size, so the information conveyed by a spike is primarily in the identity of the neuron generating the spike and its time of arrival. These stereotypical spikes act as the communication mechanism for the brain. We constantly take inputs from our senses, recognise familiar objects and learn new associations. These inputs are sequences of spikes in time and space.

In this dissertation, we focus on modelling a particular function of the brain, that of sequence learning. Sequence learning is a phenomenon that is ongoing in mammalian brains almost all the time: we learn and remember sequences of sights (such as moving images), sounds (such as songs), events, names, numbers, etc. In this work, we have developed a model of on-line sequence learning and implemented it using spiking neurons. The model can learn a new sequence of symbols in a single presentation and can complete previously learnt sequences by predicting the next symbols in the sequence. We model the engineering aspects of information transmission in a dynamical system built out of asynchronous spiking neurons, in which the symbols are encoded as spatio-temporal bursts of spikes

emitted by layers of neurons (not to be confused with the term 'burst' used in neurophysiology referring to a number of spikes emitted from a single neuron in a short time interval).

1.1 Motivation

The primary motivation for this research is to gain a better understanding of how a robust neural system can be engineered through low-level asynchronous components, by taking a concrete example of a sequence machine built using spiking neurons. It is intended to contribute to the understanding of the dynamics of interactions between artificial spiking neurons in a complex system, and give some ideas as to how biological neurons also might interact. Although part of our motivation is biology, we deal with the problem strictly as an engineering task. Another motivation is to implement the spiking neural system in custom-built hardware [68].

There have been many previous models that can perform sequence learning, some of which use spiking neurons, including work by Berthouze et al [12]. Details of these models are described in chapter 2. However, these models assume that the generative model of the sequences is fixed, and the task of the system is to learn the generative model after many trials and on this basis predict the sequence. We, on the other hand, have concentrated on the problem of one-shot learning with no assumptions being made about the model generating the sequences. We consider this problem to be more biologically realistic than those problems that the other models attempt to solve. It can also be beneficial in a number of applications in various areas such as visual motion processing, speech, robotics, some of which will be discussed in detail in chapter 9. Another difference of our approach from these approaches is that we deal with the problem at different levels of abstraction, starting from a high-level solution (using an associative memory) and coming down to a low-level, spiking neural implementation that exactly implements the high-level abstraction, thus implementing our goal of engineering a high-level system using low-level neural components. The other models have dealt with the problem of sequence learning using spiking neurons directly. There are also differences with other models in the specific coding scheme (rank order coding [77]) and associative memory architecture (N-of-M SDM [27]) that we have used in our model.

This work can be thought as a proof of concept, showing that a functioning on-line sequence machine that does not assume the model generating the sequences can be engineered out of spiking neurons. The scope of this work is multidisciplinary, and we seek to take inspiration from biology and utilise knowledge from diverse fields such as computational neuroscience, neural associative memories, asynchronous logic, etc. to build our model.

Developing functional models of parts of the brain is essential to neuroscientists, as it helps them to make sense of the large mass of data gathered from experiments, and also to get a better understanding of how the brain works. We hope that this work will prove useful to both hardware and software neural modellers, as it highlights the problems they are likely to face, and offers some solutions.

1.2 Aim

The primary aim of the work described in this dissertation is to design an on-line sequence machine using spiking neurons that does not make any assumptions about the generative model of the sequences (unlike similar models such as by Berthouze et al [12]), and uses an explicit context neural layer with feedback to store the context of the sequence dynamically. Our model uses rank order codes (first used by Thorpe [77]) as the encoding scheme and a version of Kanerva's Sparse Distributed Memory [45] using ordered N-of-M codes [27] as the associative memory.

By spiking neurons we refer to point neurons (with no spatial characteristics) which make a decision to fire based on only local input spikes, the only information transmitted by the spikes being the time of firing. In accomplishing this task, we seek to study the dynamics of propagation of bursts of spikes in complex neural systems with feedback, such as the coherence and stability of the propagated symbols. We aim to build a suitable associative memory using spiking neurons to be used in the sequence machine, and a suitable way to encode the context or history of the sequence. We also seek ways of abstracting out the temporal behaviour of a spiking system into a suitable coding scheme that is convenient to use.

One thing we would like to clarify is that we have approached this problem from an engineering perspective, which is to build a system that does what it is

supposed to do, and does it well. Although our sequence memory has features that model biological memories, we do not claim that this is the exact way our brains handle sequences or that the spiking neural model we have used can capture the huge variety of neural behaviour or the range of interactions between biological neurons. Our primary motivation is to highlight issues in modelling high-level systems using spiking neurons, and we have chosen the particular task as an interesting and relevant example of the modelling. For the same reason we do not claim our model to be the best possible way sequences can be learnt, although we have tried to optimise our model and shown that it performs the sequence learning task quite efficiently.

1.3 Approach

In this work, we have built the sequence machine using a top-down approach, starting with the high-level problem and then implementing its functionality using low-level components. In our model, we have two levels of sequences: the higher level of sequences of symbols and the lower level of spatio-temporal sequences of spikes which constitute the symbols. We have an associative memory, also formed out of spiking neurons, which can learn associations between the lower-level spatio-temporal spike sequences that form symbols.

A neural associative memory can store associations between encoded symbols, and can be read from and written to in a single pass. We build an associative memory using rank-ordered N-of-M codes as the encoding scheme codes to use in our on-line sequence machine. The type of memory we have chosen is a variant of Kanerva's sparse distributed memory (SDM) [45] using N-of-M codes [27] with rank-ordering. The learning rule used in the memory, based on setting the connection weights to the maximum of the old weights and the outer product of the two vectors to be associated, is similar to the learning rule used in binary correlation matrix memories, that is based on local information rather than error signals propagated by an external teacher. We investigate the suitability of two spiking neuron models to implement our system, namely the rate-driven leaky integrate and fire model and the wheel or firefly model [46]. In implementing the memory out of spiking neurons, modelling issues such as coherence, stability and learning are studied in detail in chapter 6.

In building the sequence machine in this way, there are multiple levels of

abstraction. For ease of modelling we have abstracted out time (which is how the spiking neurons convey information) as a rank-order code [77] and also represent a rank-order code as a vector. After that, we have analysed the efficiency of rank-ordered coding in sparse distributed memories, implemented by using this abstraction of the temporal behaviour. Properties of these memories such as information density, scalability and error tolerance have been studied. Finally, the developed optimum memory has been used to build a sequence machine, which we then implement using spiking neurons and perform tests to measure its performance. We also justified the temporal abstraction used earlier by showing that equivalent results are obtained using the abstraction and using a system built out of spiking neurons.

Concepts that the reader may be unfamiliar with, such as sparse distributed memory, N-of-M coding etc [27]. are explained in detail in chapter 3.

1.3.1 Levels of organisation

The work described in this dissertation is primarily concerned with neural modelling. We model a predictive on-line sequence machine using spiking neurons as building blocks. It is useful to think of any modelling task in terms of levels of organisation or abstraction, which reduces unnecessary complexity and enables one to focus on the task at hand. Abstraction keeps the different levels of complexity separate from each other so that we can relate to each level in a convenient way, yet maintain an overview of how the levels relate to each other.

The levels of organisation in the sequence machine proceed in a top-down fashion from the highest level (the sequence machine interface visualised as a blackbox, and possible applications of the machine) down to the structural level (structure of the blackbox sequence memory including an associative memory with a context layer) to the component level (including the context neural layer and the neural layers used as components to build the associative memory, i.e address decoder and data store) to the level of neural layer (which uses a coding scheme called rank-ordered [77] N-of-M [27] on its inputs and outputs) to the lowest level (model of spiking neuron). An illustration of these levels is given in figure 1.1.

Each of the levels encapsulates a different degree of abstraction, which we followed in our work. Accordingly, we approached the sequence learning problem in a top-down way as well, starting with the high level description of the problem in

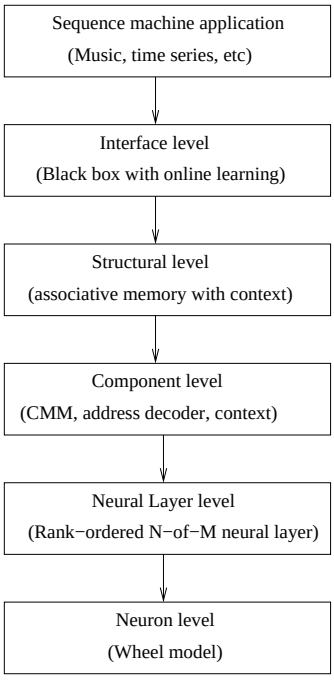

Figure 1.1: Levels of organisation in the sequence machine, from application level to the neuron level

chapter 2, followed by a description of the behaviour of the system as a blackbox in chapter 5. In chapters 4 (where we discussed the associative memory used) and 5 (where we described the context layer) we described the components of the system to implement the blackbox behaviour. In the following chapters we came down to the lowest levels, first describing a model of spiking neuron that can implement the coding scheme in chapter 6, and then describing the implementation of the system using spiking neurons in chapter 7.

1.4 Research issues and questions

In the course of this dissertation, we seek to highlight relevant issues and provide solutions to a number of problems, some of which are related to each other. Some research questions that this work aims to address are the following:

1. What are the different ways in which the performance of an associative neural memory can be measured?

2. What is a useful way to represent the concept of 'distance' or similarity between symbols encoded as vectors (which in turn represents a sequence

of firing times by a neural layer) in the system?

3. In a given sequence, how can the entire past history or context of the sequence be encoded in the best possible way in a vector of finite length?

4. How can we build a system where the spike firing times in a layer of neurons are represented and stored spatially (i.e. in the connections between neurons) in such a way that we can learn the firing order and reproduce it when needed?

5. What kind of spiking neural model should we choose for implementing specific networks, and why?

6. What are the factors influencing the stability of a burst of spikes passing through neural layers and what are the principles of designing a system to be more stable?

7. How best can we characterise the performance of a sequence memory?

1.5 Contributions

The **key contribution** of this research is that it implements an on-line predictive **sequence machine** with **spiking neurons** and uses **rank order coding** as the coding scheme. To the best of our knowledge, these three concepts of rank order coding, spiking neurons and sequence memory have never been used together in any work prior to this.

The contributions made in the three different areas mentioned in the key contribution are listed below.

Contributions in the area of rank order coding

- Using rank order coding (first used by Thorpe [77]) in a spiking neural system implementing a sequence memory, constructed using an implemention of Kanerva's sparse distributed memory (SDM) [45] with N-of-M coding and real valued weights. (See Chapter 7)

- Implementing a rank-ordered N-of-M SDM associative and a study of its performance with regards to its capacity, error tolerance and behaviour at different sensitivity values (see Chapter 4). This extends earlier work on N-of-M SDMs using unordered or binary codes, by Furber et al [27].

Contributions in the area of spiking neurons

- An understanding of issues involved in modelling a high-level engineering task (a sequence machine) using low-level spiking neurons (see Chapters 6 and 7).

- A novel spiking neural model, called Rate Driven Leaky Integrate and Fire or RDLIF, which has the activation driving function increased by incoming spikes and activation decay with time in the absence of spikes, a kernel function similar to Gerstner's Spike Response Model ([28]) and is a modification of the commonly used Leaky Integrate and Fire model [53], proposed in Chapter 6, and showing how this model is not suitable for implementing the abstraction of firing times as a vector of significances mentioned above (see Appendix A).

- Demonstrating that a spiking neural system operating in real time performs equivalently to a system implemented using a rank order abstraction of firing times (see Chapter 7).

Contributions in the area of sequence learning

- A novel framework for the asynchronous updating of the context during on-line predictive sequence learning where no assumptions are made about the generating model of the sequences, and its use to build an on-line sequence memory (see Chapter 5, section 1).

- A "combined model" of context encoding, combining the features of a neural layer (such as those used in Elman [23] and Jordan [43] models) and shift register (used in time delay neural network models [51], [83]) by means of a scaling parameter λ to modulate the influence of the context relative to the input in deciding the next prediction and to switch between the two behaviours. (see Chapter 5)

- Tests on the properties of the implemented on-line, predictive sequence memory, varying different parameters including sequence lengths and alphabet sizes and finding optimal values for scaling parameter λ for different distributions (see Chapter 8).

1.6 Dissertation structure

The dissertation is divided into nine chapters, including this introductory chapter. A summary of the remaining chapters is presented below.

Chapters 2 and 3 are literature review chapters. Chapter 2 defines key concepts and gives an introduction to the problem of sequence learning. It also provides the biological motivation and surveys various previous approaches to the problem. Chapter 3 provides an introduction to models of spiking neurons and associative memories including sparse distributed memories.

Chapter 4 is the first novel contribution of this dissertation. It describes the associative memory used in our model, which is a version of sparse associative memory [45] using N-of-M codes [27], adapted with real valued weights to facilitate later spiking implementation and is capable of reading and writing rank order codes. It presents the results of some experiments to measure the performance of the memory.

Chapter 5 is the second novel contribution made in this dissertation. It describes a framework for asynchronous updating of context in the sequence machine. It then implements a sequence machine in accordance with this framework, using a new combined model of context encoding and using the associative memory developed in Chapter 4.

Chapter 6 is the third novel contribution of the dissertation. It describes a new model of spiking neuron with first order dynamics, called rate driven leaky integrate and fire (RDLIF) model, that is suitable for implementing rank order codes. It also presents a simpler linear model called the wheel model that we have used to build our system. The chapter also discusses different issues in modelling through spiking neurons, such as the use of feed-forward and feedback inhibition, and shows through simulation how a spike burst could be transmitted stably through different layers of a feed-forward neural network.

Chapter 7 discusses the different components of the sequence machine and how each of them can be implemented through spiking neurons. It also discusses timing and other issues in the integration of all these components. The system is simulated using a suitable spiking neural simulator. It is demonstrated that the spiking system can learn a given sequence correctly and in a single pass, while observing all the timing constraints.

Chapter 8 presents the results of different experiments performed on the sequence machine in order to understand how its performance varies with different

parameters used.

Chapter 9 concludes and summarises the entire dissertation and gives pointers to directions in various related areas in which further work might be pursued. It also looks at some applications in which the model might be utilised.

1.7 Published work

Different aspects of this work have been published so far in the following publications:

- "A neural network for prediction of temporal sequences of patterns", by J. Bose, published in proceedings of EPSRC Postgraduate Research Conference (PREP 2005), Lancaster, UK, 30 Mar-1 April 2005.
- "A system for transmitting a coherent burst of activity through a network of spiking neurons", by J. Bose, S. B. Furber and J. L. Shapiro, published in proceedings of 16th Italian workshop on Neural nets (WIRN 2005), Vietri sul Mare, Italy, 8-11 June 2005.
- "An associative memory for the on-line recognition and prediction of temporal sequences", by J. Bose, S. B. Furber and J. L. Shapiro, published in proceedings of International Joint Conference of Neural Networks (IJCNN 2005), Montreal, Canada, 31 July - 4 August 2005.
- "A sequence machine built with an asynchronous spiking neural network", by J. Bose, published in proceedings of 17th UK Asynchronous Forum, Southampton, UK, 5-6 September 2005.
- "A spiking neural sparse distributed memory implementation for learning and predicting temporal sequences", by J. Bose, S. B. Furber and J. L. Shapiro, published in proceedings of International Conference on Artificial Neural Networks (ICANN 2005), Warsaw, Poland, 11-15 September 2005.
- "An asynchronous spiking neural network which can learn temporal sequences", by J. Bose, S. B. Furber and M. Cumpstey, published in proceedings of 18th UK Asynchronous Forum, Newcastle, UK, 4-5 September 2006.

- "Sparse Distributed Memory using Rank-Order Neural Codes", by S. B. Furber, G. Brown, J. Bose, M. Cumpstey, P. Marshall and J. L. Shapiro, published in IEEE Transactions on Neural Networks, Vol 18, Issue 3, May 2007.

Chapter 2

Sequences and sequence learning

In the first chapter we gave a general introduction to the thesis, and mentioned that the specific goal of our research is to build an on-line sequence machine that can do sequence prediction and can be implemented using spiking neurons. In this chapter we define and discuss the problem of sequence learning in detail, including various machine learning approaches to the problem. We shall detail the spiking neural implementation of the machine with a suitable hetero-associative memory in later chapters.

2.1 Defining some terms

In this section, we define a few terms we use throughout the thesis.

2.1.1 Defining a sequence

We define a sequence in our system as a temporally ordered list of symbols from a finite alphabet. The symbols constituting a sequence can be anything from characters or notes of a tune to stock market data. If we use symbols from the English alphabet, 'abc' is an example of a sequence of three symbols, of which the symbol 'a' is first in time order, followed by the symbols 'b' and 'c'. Since time can only go in the forward direction, we do not treat the ordered sequence 'ab' the same as the ordered sequence 'ba'.

We take the entirety of the symbols presented to the system from the start till present as constituting the whole sequence. We are building a predictive model based on the past inputs to the sequence machine, which means that the system

predicts what the next symbol should be based on the entire sequence of symbols that has been presented to it so far. Therefore, a sequence in our system is simply all the symbols that are explicitly presented to the system from the beginning.

2.1.2 The sequence learning problem

Learning a sequence means learning the temporal relationship between the constituents of the sequence. For example, if a sequence of letters 'abcde' is learnt by the machine, the machine will know that 'b' follows 'a', 'c' follows 'ab', 'd' follows 'abc' and so on.

The problem of learning sequences is harder than learning single associations, such as associating symbols 'a' and 'b'. To learn a sequence, a neural network must have some way of representing and storing the temporal information of the sequence spatially, in the connection weights. To learn a sequence such as 'abc', a machine must learn to associate 'b' to 'a' and 'c' to 'ab'. Thus, the context of any symbol in a sequence, representing where the symbol occurs relative to other symbols presented so far, must also be learnt. The memory has a finite size, therefore we need to store the context in a suitably compressed way.

The way we can verify whether the system has successfully learnt a sequence is through prediction: if we enter the symbols of the previously learnt sequence to the system one by one, it can successfully predict the next symbols in the sequence based on what it has learnt. A brute force method to solving the problem of predictive sequence learning would be to store the entire past history of the presented inputs. During recall, as we give it the symbols of a given sequence it has seen and stored previously, the system narrows down the search space of possible candidate sequences to which the given input symbols may belong, until it has just enough information to identify the sequence and if possible, unambiguously predict the next symbols correctly. There may be cases where unambiguous prediction of the symbol may not be possible. In case of such conflicts, the system should be able to predict the most likely (as per the prior probabilities, also known as Base rate) next symbol.

2.1.3 Defining a sequence machine

A sequence machine is a system that can learn sequences of symbols.

Symbols can be made up of images, sound, music notes, numbers, etc. We

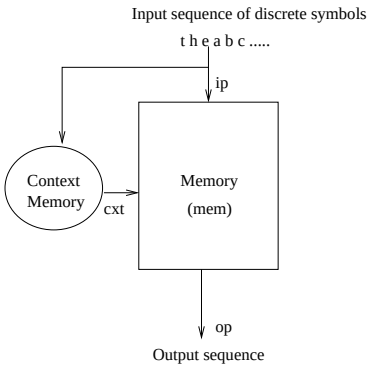

Figure 2.1: Basic design of a sequence machine

assume here that we have a finite alphabet and the sequences are formed of symbols from this alphabet. We are interested in on-line sequence learning, which means that there is no separation between training the system to learn new sequences and recall of a previously learnt sequence on presenting a few symbols as cue, and a single presentation of the sequence is sufficient for learning.

In sequence memories, the context of a symbol is important in recognising a sequence, as the same symbol can appear in different contexts. For example, two different sequences ABCDE and ZBCXF have the symbols BC in common, so the successor of C cannot be predicted unless its context in the sequence is also known. So the machine must have some mechanism to store the history or context of the sequence. This can be in the form of an explicit neural memory to store the context in the neural connections, a dynamic memory where the context is stored as the activity pattern of a neural layer with feedback, or an implicit way in which the context is built into the structure of the memory (such as by using delayed connections, an activity trace, a convolution or some other mechanism). We have chosen to use a dynamic context using a neural layer with feedback. Also, it must have a memory that associates a symbol with its successor, so that on being given an input symbol, it can make a prediction of what the next symbol should be. For this purpose we have chosen an associative memory.

Therefore, every neural sequence machine must have the following components: an input symbol, an encoder to encode the symbol into a suitable distributed neural code for storing in the memory, a memory to store the associations between encoded symbols, a memory to store the context of the symbol, a decoder and the predicted output symbol. Figure 2.1 shows the structure of a basic sequence memory. The structure of the sequence machine we have designed

will be explained in more detail in chapter 5.

2.1.4 Defining a symbol and a sequence in a spiking neural system

Our intention is to build our system using spiking neurons as components. Accordingly, the symbols constituting the sequences are encoded as spike bursts emitted by layers of spiking neurons firing asynchronously. We need first to define a symbol in terms of the fired spikes. Here we define a symbol to be encoded in the choice and temporal order of firing of spikes emitted by a layer of neurons. The encoding using the time order (or rank) of firing is called rank-order coding and was first used by Thorpe [77]. The number of firing neurons out of the total number of neurons in the layer might be fixed, in what is called N-of-M encoding, in which the choice of N firing neurons in a firing burst out of a total of M neurons in the layer is used to encode the symbol [27]. We have used an ordered N-of-M encoding as the scheme for encoding symbols in our system, as a combination of N-of-M encoding and rank-order codes.

Decoding is the reverse of the encoding process. A particular sequence of firings of the neurons in a layer is decoded as a symbol by identifying the encoded symbol that is closest or most similar to it. We have a fixed alphabet of symbols and the number of symbols in our alphabet is less than the maximum possible number of codes (combinations of neural firings that make up our symbols). Therefore we can afford some degree of redundancy in identifying a spike sequence as a symbol in the alphabet. We have also defined a way to measure similarity between symbols, in order to identify the output sequence of spikes with one of the symbols in our alphabet.

The encoding and decoding schemes will be explained in more detail in chapter 4.

In implementing the sequence machine using spiking neurons, there are two levels of temporal sequences in the system: the sequence of spikes output by a layer of neurons forming a burst which encodes a symbol and the sequence of symbols. In this and later chapters where we refer to the term ‘sequence’ we will take it to mean the sequence of symbols only.

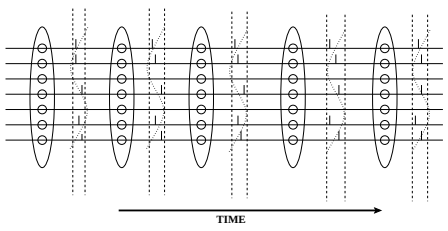

Figure 2.2: A spike wave front propagating through a series of neural layers in time, representing symbols propagating across different layers. The ellipses represent layers of neurons, and the bursts of spikes emitted by each layer (between the dotted parallel lines) are shown between the layers in the shape of a wave (represented in the dotted S-shaped curve) travelling forward in time. This figure illustrates what we mean by the terms 'burst' and 'wave'.

2.1.5 Defining a wave and burst

The spiking activities in the system occur in waves. First, an input spike is fired, which causes a wave of corresponding spikes to fire in all the other layers of the system, until the output spikes are fired, similar to the functioning of a synfire chain [1]. This is followed by the next wave and so on. A so-called wave is propagated by successive layers in a system firing bursts of spikes (forming a rank-ordered N-of-M code) into the next layer in a feed-forward fashion, which in turn fires another burst of spikes, and so on, as shown in figure 2.2. In this and successive chapters, we will use the term 'burst' to refer to the spikes emitted by a neural layer forming an ordered N-of-M code, and 'wave' to refer to a series of successive bursts in different layers of the system triggered by an input spike.

2.1.6 Defining neural and non-neural

In this dissertation, we shall use the term 'neural model' to mean local point units or nodes arranged in a weighted directed graph structure (with weighted connections between the nodes, which can be both forward and recurrent), each node having a quantity called activation that is incremented when it gets inputs from other nodes. If the activation of a node exceeds a threshold, it is said to have fired and the activations of all the nodes connected to it are incremented. All models that do not consist of nodes with weighed connections and activation shall be referred to as 'non neural' models.

2.2 Problem formulation

In this section, we define our sequence learning problem precisely.

In a sentence, we can state that our problem is to build a context neural layer based predictive model of on-line sequence learning with a finite memory that can be implemented using asynchronous point spiking neurons and does not make any assumptions about the generative model of the sequence. We shall now describe in detail what this means.

We want to build a system that can learn any given sequence, which it has not learnt before, on a single presentation of that sequence, and predict the next characters of a previously learnt sequence as accurately as possible. The system has a finite memory to which associations are written and from which they are read. The learning is on-line, which means that the machine works in one mode only and there is no separation between the learning (write) and recall (read) modes. When we speak of a predictive sequence machine, we can imagine a black-box such that we enter the symbols constituting the sequence one by one, and the machine learns the relation between the symbols. On every successive symbol that is entered, the machine gives a prediction of the next symbol (after reading from the memory). The behaviour of the machine in the **ideal case** is as follows: If the symbol is part of a sequence that has been previously learnt by the machine, the prediction will be correct and no new learning will take place. If the symbol is not part of a previously learnt sequence, the prediction will be a random symbol but the machine will write the new association of the old context and the new symbol to the memory, and predict it correctly the next time the same sequence is presented. Thus the learning and prediction are integrated, and the machine knows when to learn a new association.

The problem we have in mind is getting the best prediction of the next symbol in the sequence based on the associations written earlier to the memory. The prediction of the next symbol depends on the context of the present symbol in the sequence as well as the current symbol presented. The machine should be able to disambiguate the next symbol and predict it correctly, with as few input symbols as necessary for it to reconstruct the context and identify the next symbol uniquely. The context should be more influenced by recent inputs than by the past history. Yet it must not forget history completely, to enable it to disambiguate between two very similar looking sequences.

The system learns the association between the old context and the current

symbol, and based on this the new symbol is predicted. If a similar looking context and the same symbol are presented to the machine as an association that was learnt previously, it would be able to predict the next symbol correctly. In an ideal case with infinite memory, the system should be able to look back in its memory from the current state as far as necessary to find the association between the closest looking context and current symbol that was learnt previously, and predict the unambiguous next symbol accordingly. In case of ambiguities, the system should give preference to the association that was learnt more recently. This resembles features found in a human memory, since we generally tend to remember sequences learnt more recently and forget those we learnt long time ago. However, this approach would fail where subsequences are repeated as part of a bigger sequence, for example in the sequence 'abcabdabcabd...', where the repeated subsequence is 'abcabd'.

Since the machine associates the context with the current input, it should be noted that it cannot concatenate sequences automatically. For example, giving it the sequences 'abc' and 'cde' separately at different times does not mean that the machine has automatically learnt the sequence 'abcde'. It may or may not predict 'd' for 'abc' depending on the context at that time.

We now describe the behaviour of the sequence machine with an example, assuming an infinite memory and no errors in the presented sequence. Suppose we have a sequence much as 'abcabdab'. The machine has to make the best prediction for the next input symbol (the successor of the last 'b') based on what it has seen so far. The way it can do so is this:

1. The machine scans the whole sequence from the start to see the instances where this symbol has occurred earlier in the sequence. In this case, it will scan for instances of 'b'. If there is only one other instance, it can state the prediction based on that instance. Since 'b' is followed by 'd' and 'c' at different times, we cannot determine the successor based on this alone.

2. If we cannot deduce the unambiguous prediction based on step 1 alone, we increase the length of the window until we can find an earlier instance of the same sequence. For example, if the length of the window is 2, it has seen 'bc' and 'bd'. But here too there are two instances and each is followed by a different character. Therefore we increase the length of the time window to 3, and here too we have 'abc' and 'abd' so here too the unambiguous prediction of the successor of 'b' cannot be determined.

3. In instances like above where we cannot develop an unambiguous prediction, the more recently presented sequences will take precedence. For example since 'abd' was the more recently presented subsequence among the two possible predictions 'c' and 'd', the machine will predict 'd' as the successor of 'b'.

The above example illustrates how some ambiguous cases of prediction can be resolved. The example assumed that we had no errors and the memory to store the sequence was infinite. If, however, we implement the machine using a neural hetero-associative memory, where the entire sequence until now is treated as the cue and the prediction of the next symbol is based on this cue, there is the possibility of errors in the sequence such as missing characters. In case of missing symbols in the larger sequence, a neural memory would still be able to make the best guess although the above algorithm would still be ambiguous.

2.2.1 Extension to the problem formulation

In recalling a previously learnt sequence, we make an extension to the basic model by stipulating that the machine should be able to work with errors in the cue which is used to recall the previously learnt sequence. In the case of such errors such as incomplete cues, the machine should be able to predict the next symbols on being given as few new inputs as possible i.e. it should be able to reconstruct the original context by 'closing in' to the context as more characters are input. For example, if the sequence 'abcr' is learnt, and the symbols given after learning are 'bc', it should be able to deduce that 'bc' is a part of the old context 'abc' (in the absence of an alternative learnt association) and therefore the predicted symbol should be 'r'. Also, the context formed by 'bc' should get closer to the original context as more symbols are added. For example the context formed by 'bcde' should be closer to the context formed by 'abcde' than 'bc' is to 'abc'. This would enable us to remember any sequence from the middle as well, and thus make the system more tolerant of errors in the sequence. In case of such mistakes in the recall cue, the system should be able to lock on to the correct sequence and hence predict correctly on being given a minimum number of additional inputs.

2.2.2 Using a hetero-associative memory implemented in spiking neurons

Since we intend to implement this system using spiking neurons, we want a neural equivalent of this ideal case, that has the added features a neural memory can bring, such as robustness to failure of some components. The type of neural network we have used for this task is a hetero-associative memory. The memory associates a symbol with another, and the stored association can be retrieved using one of the symbols as a cue. As described in the previous chapter, associative memories learn associations, and they can do it in a single pass or single training cycle, i.e. on a single presentation of the two symbols to be associated. This one-pass learning is suitable for our requirement of on-line memories, and this is why we chose associative memories for the task.

Using a neural associative memory presents a few additional constraints: the number of sequences the system can learn is limited by the dimensions of the memory. Since the size of the memory is finite, the system will inevitably have errors in the prediction as the memory fills up. Also, the neural network has no way of knowing what it knows, unless it is explicitly programmed for that purpose (for example, by defining a way to measure similarity or distance between encodings, and then defining a threshold of similarity which is the confidence level above which the system determines that it has recognised the sequence the symbol belonged to).

2.2.3 Some notes on the functionality of the high-level system

In our model, we treat the sequence as a continuous long chain of symbols, with each learnt sequence such as 'abc' being a part (or sub-sequence) of this long sequence. There is no way in the system to separate sub-sequences (although the blank character can be a part of the alphabet) or to indicate the beginning or end of a sequence such as 'abc'. This is intuitive to the way humans learn sequences, because we do not have any memory from the beginning and recall previously learnt sequences based on the totality of the history we have learnt so far. If we wish to learn separate sub-sequences, we could have a simple extension to the system to incorporate this feature, by simply clearing the context when such a special "end" character is input. The system we are implementing, however, will

deal with a continuous chain of symbols without any special characters.

2.3 Motivation: Biological neurons, networks and spikes

The brain is made up of cells known as neurons. The human brain contains more than 10^{11} neurons, according to the Scientific American Book of the Brain [3]. It also has around 10^{15} synapses (connections between neurons).

Biological neurons generally have a similar structure although with variable morphology, consisting of a number of dendrites, a cell body called the soma and one or more long axons. The shape of a neuron is shown in figure 2.3.

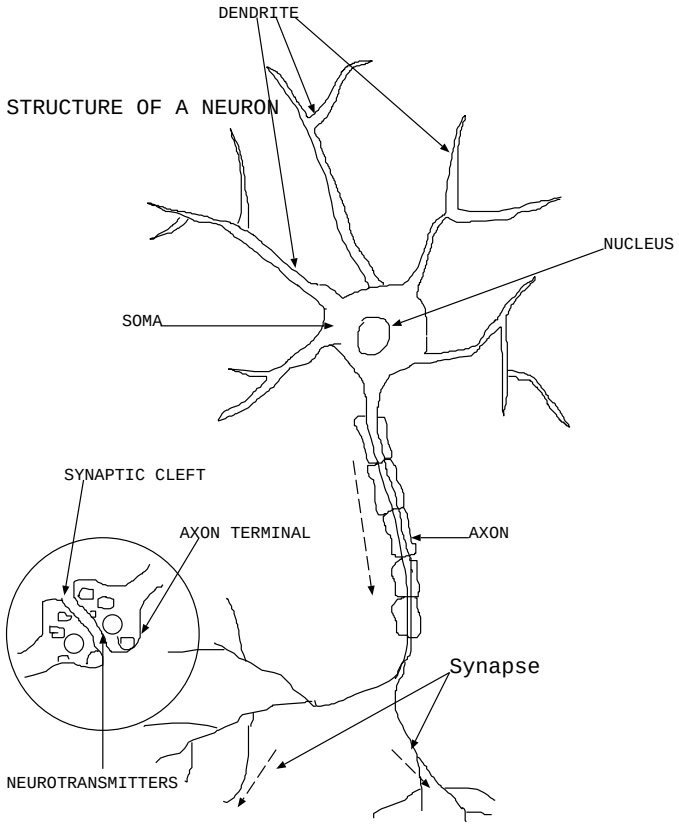

Figure 2.3: Structure of a neuron (adapted from Longstaff [52])

Information is transmitted between neurons through electrical impulses called action potentials, also referred to as spikes. The dendrites connect to other neurons and take input spikes from them. Output spikes generated by a neuron travel through its long axon to be transmitted to the dendrites of the neurons

connected to it through the synaptic cleft, which is the name given to the gap between neurons.

There is great diversity in the behaviour of spikes emitted by neurons, such as bursting or spiking in phases or without phase, spiking in response to different kinds of inputs in different ways, spiking with different response times, etc. The type of spiking behaviour varies with the type of the neuron, its location and the type of input it receives. The time frame for the firing of a spike is of the order of milliseconds.

2.3.1 Mechanism of spike production

Action potentials or spikes are electrical impulses caused by electrochemical mechanisms and causing the transfer of chemical messengers called neurotransmitters from the axon of one neuron to the dendrite of another. They change the electrical potential of neurons through the transfer of various ions (mainly potassium, sodium, calcium and chlorine) due to the concentration differences of these ions across the cell membrane of the neuron, and the resulting difference of electrical potential. Neurotransmitters released from the axon of one neuron travel across the synaptic cleft and bind to receptors in the dendrites of the connecting neuron, causing its electrical potential to change which may lead to the generation of an output spike by that neuron.

Neurons at rest have a specific negative electrical potential in the cell body called the resting potential, which is of the order of a few tens of millivolts. Incoming spikes can increase or decrease this potential. Presynaptic spikes (spike transmitted from the synapse of a neuron to another receiving neuron) can be either excitatory or inhibitory. When an excitatory presynaptic spike arrives at the synaptic cleft, it causes a depolarisation of the synaptic terminal, i.e. the resting potential becomes less negative. This induces neurotransmitter molecules to be released, which change the membrane potential (also called cell potential) of the postsynaptic neuron (the neuron which receives the neurotransmitters) by causing positive sodium ions to be released through voltage-gated ion channels. If the membrane potential happens to exceed a threshold, a spike is emitted, marked by a sharp, transient increase in the membrane potential, which is known as depolarisation. Depolarisation is followed by inactivation of the sodium channel and the opening of the potassium channel, and the membrane potential goes

down again, below the resting potential of the neuron, in what is termed hyperpolarisation. Hyperpolarisation causes deactivation of the ion channels and a return to the resting potential of the cell and this completes the spike. Spikes generated at the soma are transmitted across the axon of the neuron at a constant speed which varies with the type of neuron, ranging from 2 to 400 km per hour. These mechanisms can be studied in more detail in books such as "Principles of Neural Science" by Kandel, Schwartz and Jessell [44].

The moment when the membrane potential crosses the threshold from below (during depolarisation) is termed the firing time of the spike. After the spike firing time, when the membrane potential is reset below the resting potential, the neuron is unable to fire another spike for some time, which is known as the refractory period of the neuron.

A neuron can have either inhibitory or excitatory outputs, but cannot have both (Dale's law, [18]). In biological neurons, excitatory synapses are more numerous, but inhibitory synapses are located closer to the cell body, are stronger and typically work on faster time scales.

2.4 Sequence learning in biology

Most organisms in nature have to learn, remember and predict sequences. For example, we constantly remember and recall sequences of sights, sounds, events, etc. The human memory is capable of sequence learning. As babies, we know almost nothing about the world. As we grow up, we learn all kinds of sequences that help us form a picture of the world, including phone numbers, names etc. We learn things by association: we associate people with events, events with other events, etc. In this section, we examine the relevance of the sequence learning problem by reviewing a variety of such biological phenomena related to sequence learning that have been studied and modelled.

2.4.1 Working memory

Baddeley [8] et al postulated models of human memory, including short-term, medium-term (also called working memory) and long-term memory. They proposed that as a sequence gets repeated, it gradually shifts from dynamical short-term memory to long-term memory. According to this model, there are three main components of human working memory: the phonological loop (which acts

as a rehearsal system for words), visuo-spatial sketch pad (which is responsible for things such as remembering spatial orientation and interfacing between visual and spatial elements of our sensory and motor organs) and the central executive (which interfaces between these systems and also performs attention switching). This model postulated the existence of some way to store the context or history in a sequence, but did not specify its exact mechanism.

2.4.2 Episodic memory

Episodic memory, which is responsible for remembering of episodes by the brain, is another biological phenomenon involving the learning of sequences, since episodes are connected sequences of sights, sounds, etc that are remembered and recalled as a whole. The hippocampus is the part of the brain involved with episodic memory. Shastri [69] developed a model of episodic memory in the hippocampus, in which short-term memory represented by a transient pattern of activity gets transformed into long-term memory as a persistent memory trace by long-term potentiation.

2.4.3 Sequence learning and prediction in the neocortex

There is a theory, recently put forward by Hawkins in his book "On Intelligence" [31], that the neocortex (a part of the mammalian forebrain) acts as an on-line sequence learner and predictor. This theory is based on observations by Mountcastle [56] that parts of the brain dealing with processing of different senses should have similar principles. Our senses such as the eye, ear etc constantly receive inputs from the outside world, which are then converted into spike trains. As per the theory, the higher layers in the hierarchical neocortex form predictions of the world, which are propagated below and matched with the sensory inputs at the lower layers, ending with the sense receptors such as the eyes. When a prediction in a layer matches the sensory input, the input is identified and a label is propagated to the higher layers, thus reducing unnecessary details at the higher layers. If the prediction does not match, a more detailed representation is propagated, causing the higher layers to form new labels if necessary and generate new predictions, such that the predictions of similar sensory inputs will be correct in the future.

For example, if we see an object on our desk one morning that we are not used

to seeing regularly, say a new vase, we immediately take notice because it does not match our prediction based on familiarity of how our desk should look like. After some time we get used to the new information and when we see the vase again, we do not notice it since it agrees with our visual prediction. Hawkins claimed that this memory-prediction framework was basis of intelligence. However, it should be noted that this theory is highly speculative, nevertheless it is an interesting model of how predictive sequence learning might be of use in the brain, which is similar to the problem we are considering.

2.4.4 Sequence learning in songbirds

Another interesting biological phenomena involving sequence learning that has been studied and modelled is song learning and recall in young songbirds. Troyer et al have studied how songbirds such as male Zebra Finches learn and reproduce sequences of sounds [80, 81]. They found that young songbirds repeat a heard song to themselves until they can reproduce it correctly and learn the song syllable sequence as a chain of sensory and motor representations. They identified different areas of the songbird brain such as the anterior forebrain pathway (AFP), high vocal centre (HVC) and robust nucleus of the archistriatum (RA) as responsible for different aspects of sequence learning. The temporal structure of the song syllables is encoded and stored in the RA region. An internal representation of the context of a syllable in a song is formed in the HVc area of the brain, and that is associated with the motor signal in the RA area by a simple plasticity-based algorithm implementing the associative memory. This learning of associations with feedback acting as teaching signals has been postulated as a model of sequence learning in the songbirds.

2.4.5 Relation of the sequence machine to the biological mechanisms

We see that in the biological models of sequence learning described above, some mechanism to store a representation of the temporal context is common. The sequence memory that we have developed uses an explicit neural layer to store the past context. In our model, the output is a prediction of the following input symbol in the sequence, as in Hawkins' memory prediction framework.

2.5 Modelling learning

This section introduces some of biochemical mechanisms proposed to model learning or the change of the behaviour (of individual neurons or the organism as a whole) over different time scales in neurons in response to inputs.

The brains of mammals and lower animals store both short-term and long-term memories. Short-term memories store dynamic activity and last for a short time ranging from milliseconds to a few hours, while long-term memories last days or longer and are caused by more permanent changes in the synaptic connection strengths. A variety of phenomena can be characterised as learning. Studies have been done on the behavioural responses of neurons in a variety of organisms such as Aplysia, which is a type of sea slug [5, 44]. Such experiments have been conducted not only by neuroscientists but also by behavioural psychologists. The phenomena studied and modelled include conditioning, starting from Pavlov's experiments [61] on dogs at the behavioural level (classical conditioning) to more recent studies of phenomena at the cellular level.

Some simple phenomena which can be considered as short-term memory are habituation (decreased response of a neuron with time with persistent application of a stimulus) and sensitivity (heightened or increased level of response of a neuron to a new input stimulus). Conditioning, in which an organism can be trained or conditioned to respond in a certain way to a given stimulus, is another example of learning.

Sometimes long-term learning takes place, causing the strength of the synaptic connection to be increased or decreased. Hebb's rule [33] is an important localised learning principle for the long-term change of the connection strengths between neurons. It states that if a neuron persistently takes part in the firing of another, the connection strength between them is increased. Hebbian leaning requires both temporal correlation (both the neurons should fire close to each other in time) and spatial proximity for the connection strength change to be possible. In recent times, biologists have found evidence of this mechanism in real neurons. Two phenomena are commonly cited as evidence: Long-term potentiation (LTP) and long-term depression (LTD). LTP happens when there is a significantly correlational and perhaps causal relationship between two neurons, i.e a neuron 'causes' another to fire, which means that it fires just before the other one, and is characterised by an increase in the connection strength. LTD is the reverse of LTP, and refers to a decrease of connection strength when a neuron

fires just after another.

Below we briefly review some of these mechanisms, to provide a context for our work.

2.5.1 Classical conditioning and eligibility traces

Classical conditioning [5] models the higher-level behavioural aspect of learning, and is based on experiments performed by Pavlov in the 1890's [61]. In the original experiment, a dog was trained by ringing a bell (Conditioned Stimulus or CS) before it was given some food (Unconditioned Stimulus or US). After this training was repeated a few times, the dog learnt to salivate on the ringing of the bell (Conditioned Response or CR) expecting food to be provided, even when there was no food accompanying the bell.

In a variation of Pavlov's experiment, the dog learnt to salivate on the ringing of the bell although the food was only provided upon a delay after the ringing of the bell. To model this, a decaying variable called eligibility trace (to store the eligibility of learning during the delay) was proposed, and during learning the association was learnt between this eligibility trace and the response (food being provided), instead of the original stimulus (ringing the bell) and response. Therefore, the eligibility trace is a mechanism to explain delayed reward learning in situations where the phenomena to be associated do not occur at the same time (i.e. there is a time lag between the stimulus and the response, such as between the ringing of the bell and the providing of food). A recent paper [59] has shown evidence for the presence of eligibility traces at the cellular level in neurons.

In our associative memory model implemented in spiking neurons, since the spikes to be associated arrive at different times, we have used a mechanism (the significance vectors stored in the synapses of a neuron) that can be considered similar to eligibility traces. It will be explained in more detail in chapters 4 and 6.

2.5.2 STDP mechanism

Spike Time Dependent Plasticity (STDP) is a combination of LTP and LTD, which were mentioned earlier in this section. Plasticity refers to the property by which synapses are changed in the long term. STDP is a biologically plausible

mechanism to explain the Hebbian phenomena, and is more biologically accurate and realistic than the traditional Hebbian learning rule, which takes into account the correlation between pre and post synaptic spikes but misses out on the temporal aspects of it. The intention is to establish causality for learning to take place, i.e. an event happening just before another can be argued as having caused it. STDP is modelled as a relation between change of connection weight and the time difference between presynaptic and postsynaptic spikes [13].

2.6 Relation of our work to other approaches

Sequence learning is not a new topic, and a lot of work has been done previously to build neural and non-neural models to perform sequence learning. In general, the sequence learning problem can be divided into sequence generation, recognition and prediction [74]. Sequence generation refers to the generation of new sequences from a specified grammar. Recognition refers to the identification of a sequence that has been learnt previously. Prediction means the ability to predict the next symbols in a sequence learnt previously. Another common sequence learning problem is sequence completion [74], which is completing a previously learnt sequence on being given a few inputs belonging to a sequence. Sequence completion is essentially the same as sequence prediction, once the model has successfully learnt a sequence.

In this work, we concentrate primarily on building a model to perform the sequence prediction task. However, as an aside, we can argue that sequence recognition is an added functionality of our model, if we feed back the predicted characters as inputs until the whole sequence is recalled. In such a case, we consider a previously learnt sequence to be recognised if the machine can predict all the symbols accurately.

One important feature of our system should be mentioned here: the system can learn only what has been presented to it explicitly, it cannot deduce and learn higher-level relationships between symbols, such as the model that generates the sequence. Since we make no assumptions about the generative function, our model is not constrained in the type of sequence it can recognise. For example, learning a sequence such as ‘aba’ in our model is not the same as learning the sequence ‘cdc’, even though the structure of these two sequences is the same. Our memory cannot deduce the equivalence of such different relations because

it treats all encoded symbols equally (it cannot learn that 'b' comes after 'a' in the alphabet or similar relations, but can only learn that 'b' follows 'a' in the sequence that is explicitly presented to it).

Our idea of the problem of sequence prediction is different from common problems involving learning a grammar or time series prediction. In problems of grammar induction, the main task is to build an internal model of the grammar based on given training sets (which are generated by that grammar) after going through many training iterations, so that the system can predict as accurately as possible the next symbols in the sequence in the test set. In time series prediction, the problem is to predict as acurately as possible the future data in a time series, based on past data. Such problems can be solved by a system that can deduce higher-level relations between data presented to it while building its internal model of the generative function based on the training data, assuming that the data is generated by a single process and that process can somehow be learnt and modelled. On the other hand, in our model we are only interested in remembering those sequences that are presented before it, not in learning to generalise a class of sequences. The other difference in our model compared to existing predictive models is that we are building it out of spiking neurons and training it in one shot.

Related work on predictive models of sequence learning (i.e. those models which generate a prediction of the next symbol in a sequence) can be divided into neural and non-neural approaches, or into approaches which have a representation of context or some way to encode and store the past history and those that do not. Some approaches use a mixture of neural and non-neural methods, and it is difficult to classify them as belonging to an exclusive category or paradigm. It should be noted that even models in the literature that sound similar to ours, i.e. which claim to perform context based predictive on-line sequence learning, do not necessarily deal with the same problem or have the same aims. However, it is still beneficial to analyse them as some of the issues involved are very similar, and we would benefit from the insights in those approaches.

2.6.1 Sequence learning using neural networks

Sequence learning models using artificial neural networks can be classified based on the neural model used (its network architecture or learning rule), the way we represent and store the context or history, etc. Mozer [57] and Sun [74] have

proposed classification schemes and classified many of the existing models in this way. Sequence learning models need not be neural: Hidden Markov Models [9] are examples of non-neural (statistical and probabilistic) approaches to the problem.

Sequence machines have been developed for a number of applications such as music composition [58], protein sequence classification [16], robot movement [4], grammar learning [23], time series analysis (for example, stock market prediction or weather forecasting), etc. A review of existing approaches to sequence learning in a variety of domains can be seen in the book on sequence learning edited by Sun [74] and the recent paper by Worgotter et al [86].

Some of the major neural approaches to sequence learning are listed below, along with instances where they have been used:

- Reinforcement learning and temporal difference learning [75, 86]
- Traditional recurrent models such as Time delayed neural networks (TDNNs) or Elman networks [23]
- State of the art recurrent nets such as Long Short Term Memory (LSTM) [34]
- Hopfield nets ([29])
- Self organising maps [73]
- Competitive nets [4]

A brief description of some of these models follows.

Reinforcement learning and Temporal Difference learning

Reinforcement learning was based on earlier work on optimal control by Bellman in the 1950s [11] and Q-learning by Watkins et al in the 1990s [82], and reviewed in detail in a textbook by Sutton and Barto [75]. It is a model in which an agent learns from the environment, which acts as a teacher. The agent builds an internal model of the environment and uses it to interact with it through outputs called 'actions', which generate 'rewards' which the agent seeks to maximise. This differs from supervised learning in that there are no explicit 'correct' outputs which the agent can use to train itself. If we consider the environment as the generative

process of the sequence which the agent has to model as accurately as possible, this is similar to the sequence learning problem.

Temporal difference learning [79] is a model for learning in case of delayed rewards (i.e. when the feedback from the environment does not come at the same time when the action takes place, and the agent has to associate the action with the reward), using eligibility traces. As mentioned in the previous chapter, an eligibility trace is a term that represents how much the neuron is 'eligible' to learn, and is a way of remembering over time. The model can learn a sequence by associating an eligibility trace of one encoded symbol with the other encoded symbol, where the symbols to be associated arrive at different points in time.

Recurrent neural networks

Recurrent nets (artificial neural networks with feedback connections) which commonly use gradient descent to minimise errors at the output (usually with error backpropagation as the learning algorithm), have long been used as models of sequence learning. These are commonly used in supervised learning, where the network is first trained via a training set of sequences and is then used for predicting the future values in the sequence in a different test set.

Time delayed neural networks (TDNN) [32] are a model of sequence learning with a recurrent neural net. The network has three layers: the input, hidden and output layers. The input vector is composed of the present input and a series of snapshots of the values of the last few discrete inputs in the sequence. The system learns to predict the next output based on the last few inputs in the sequence. Training of the network is done with the usual gradient descent algorithm.

The Jordan [43] and Elman [23] models, which we describe later in this chapter in the section on context based models, are also based on a recurrent architecture.

Long Short Term Memory (LSTM)

Schmidhuber's method of Long Short Term memory [34] is a state of the art RNN (recurrent neural net) which can be trained very efficiently and which can remember error signals over long time lags of up to 1000 time steps. The novelty of this approach lies in having threshold gates to stop the error signals from being forgotten. The key component of this model is the LSTM cell, made up of linear and nonlinear units with gating to prevent important error signals from escaping, and irrelevant signals from entering the cell. It is thus a version of short term

memory (with recurrent connections) or a dynamic representation of the state, that could maintain the state over long time lags, hence the name. It was used for a number of applications, and shown to solve benchmark problems such as learning an embedded Reber grammar [64], noise free and noisy sequences with long time lags [34].

Reinforcement learning (using actor critic methods, where the machine adapts itself to a changing external environment by correcting its internal state based on feedback error signals from an internal critic) and gradient methods (based on algorithms similar to backpropagation to train the model using several training cycles) have been generally favoured in most of the literature, especially where practical applications are concerned.

2.6.2 Non-neural approaches to sequence learning

An important non-neural approach to the sequence learning problem is using Hidden Markov Models or HMMs [5]. HMMs have a limited number of discrete internal states. For each new input in the sequence, there is a transition from one state to another. The transition probabilities between different states are learnt from the training data. From the transition probabilities we can calculate the probability that a given sequence is generated.

Bayesian nets have been used extensively in studying time series. They are based on statistical probabilities and seek to predict the next item in the sequence with the maximum probability.

There are many models of context free and context sensitive languages for grammar generalisation according to predefined rules. The models seek to remember the rules for generating permissible sequences by the grammar or to identify if a sequence belongs to that grammar.

For prediction of a learnt sequence, tree pruning, hash tables and other search algorithms can be ways to predict the next symbol in the sequence given any symbol. The search tree or graph is built while training the system.

Fractal machines

Tino [78] described a predictive model of sequences (the problem of predicting the next symbol in a sequence given any sequence) similar to a variable length Markov model. His approach was based on the principle of holographic crystals.

He used a fractal representation of the sequence, where the entire sequence was encoded as a structure of points on a hypercube. The common feature of this approach and our work is that both are building a predictive model where we encode the context in a way that uses information from the entire sequence, although in our model the context vector has a finite length. Another difference of Tino's predictive model from ours is that Tino's model was first trained using training sets of data, and then was used to predict the next symbol, while in our case the training and prediction happen simultaneously.

2.6.3 Spiking neural models to learn spatio-temporal sequences

Recently, a few people have put forward models that can learn and reproduce spatio-temporal sequences of spikes [12, 53, 38], or perform other computations on them. It must be remembered here that our definition of sequences refers not merely to sequences of spikes, but to sequences formed by these spike sequences. In our model, every spatio-temporal spike sequence encodes a symbol, and our machine learns and predicts sequences of symbols. Below we look at some biologically plausible spiking neural models that have been proposed that are capable of sequence learning.

Cell assemblies and synfire chains

The cell assembly is a model put forward originally by Hebb [33] in 1949 as a possible mechanism of learning in the brain. A cell assembly is a group of neurons that fire together due to having strong feed-forward and recurrent connections between them, and their firing together can be thought to represent a particular concept (for example thought) in the brain dynamically or used to perform a task. The connection structure is such that these firings are largely self sustaining.

A cell assembly is formed through the application of Hebb's rule, which states that if two neurons fire in spatial (they are located close to each other) as well as temporal proximity (fire very closely in time and one of them causes the firing of the other) then some kind of physiological mechanism takes place such that the capacity of one neuron to make the other fire is strengthened. With this rule, the three conditions: spatial and temporal proximity and causality, must be fulfilled for effective learning to take place. Hebbian learning is localised, in the sense

that the neurons learn based only on information available locally to them: there is no global feedback or any other regulatory mechanism to control their firing.

A similar model to a cell assembly was proposed by Izhikevich, which he called polysynchronous groups [37], in which groups of oscillators in a network fire in synchrony.

A synfire chain can be considered a subset of a cell assembly, having only feed-forward connections between layers and no recurrent connections. Synfire chains were proposed by Abeles as a model of information transmission through cortical neurons in the brain [1]. A synfire chain is a chain of neuronal layers of comparable size with feed-forward connections, capable of sustaining and transmitting a sequence of spike firings across the neural layers. It is a logical chain, not necessarily a physical one, and the neurons constituting a chain might not have spatial proximity. A single neuron can be a member of more than one synfire chain, so such chains can store more than one spike sequence. Cell firings in one layer of the chain cause a synchronised firing activity in the next layer. Abeles also proposed some restrictions on the converging and diverging connectivity pattern of the chain in order to sustain a uniform level of firings from one layer to the next.

Sterratt [72] has studied various properties of synfire chains including training algorithms, dynamics and speed of recall and developed a model of the olfactory system based on these. Jin [41] has used a synfire chain with inhibitory and excitatory connections to build a model to recognise spatiotemporal spike sequences.

Liquid state machines and echo state networks

Interesting new paradigms have also been explored to solve the problem of computation on input spike sequences, or to extract information from fast-changing spike trains. A suitable example is Maass' approach based on the dynamics of learning waves in liquids, used in his liquid state machine (inspired by neocortical columnar memories) [54].

Liquid state machines (LSMs) were proposed by Maass [54, 5] as a biologically plausible model that could learn or perform computations on spatio-temporal sequences of spikes. An LSM has an internal representation of system state stored in a recurrent circuit, which can change continuously or in a 'liquid' manner as it gets new inputs. The model has the input, hidden and output layers, which could be built of integrate and fire neurons. The hidden layer of the model is a

recurrent nonlinear filter circuit which transforms the time varying input into a trajectory of internal states. It serves to project the input and the present state into a very high dimensional space so as to make it linearly separable. This linear separation is performed by the output layer, which is made of a linear memoryless readout map trained for a particular task. Maass showed that such a model could in principle approximate any real-time computational task on continuous input streams. Different readout units in the output layer can be trained to perform different computations on the input train, such as temporal integration of input spikes, coincidence detection, etc.

ESNs or echo state networks were proposed by Jaeger [38]. They are like a discrete version of the liquid state machines. The principle is the same as the LSM, which is having a first layer which is large and heavily recurrent and which can be seen as a reservoir of complex dynamics. The outputs are 'tapped' from the reservoir of excited signals from the recurrent layer by a trainable linear output layer. The echo state network, like the LSM, consists of three layers of neurons, with one hidden layer apart from an input and output layer. Connectivity is stochastic and weights are set in the output layer with a one-step learning rule involving linear regression.

2.7 Context-based sequence learning

There have been a few studies of context-based sequence learning, where the output of an explicit context neural layer is used as a cue for the prediction of the next symbol in the sequence [12, 23, 43].

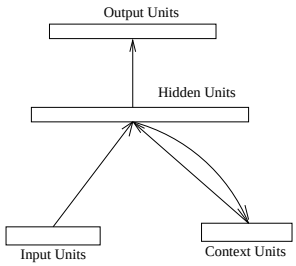

Figure 2.4: Structure of the Elman model

Jordan's model [43] used a recurrent neural network with separate input, output, hidden and context neural layers. The outputs were fed back via recurrent connections to the context layer. The output of the context layer served as a

representation of the dynamic state or history of the sequence.

Elman's model [23] was quite similar to Jordan's model, except that in this case the hidden layer output was fed back into the context layer. This model was shown to solve problems like XOR and discover syntactic features for words. This model is illustrated in figure 2.4.

In both the above models, the context layer output was fed to the hidden layer, and the hidden layer outputs were fed to the output layer via feed-forward connections. Both models had recurrent connections to the context (from the output and hidden layer respectively), and were trained using supervised learning based on minimisation of the output error. Our model is different from them in the sense that it is based on an associative memory and uses unsupervised learning. These models operate in an offline mode, involving first training the model for a number of training cycles, and then (after the training is completed) presenting incomplete parts of the sequence and expecting the model to complete them correctly.

Berthouze's model [12] had a structure similar to the above two models, with a central module (similar to a hidden layer) consisting of multiple layers of leaky integrate and fire [53] spiking neurons and four other neural layers: input, context, predicted context and output. The dimensions of the central module layers was equal to that of the input and output, and the dimension of the context was the same as that of the predicted context layer, which was not necessarily equal to the input layer dimensions. The input layer was connected with unit weights and one-to-one connections to each layer in the central module, and so was the output layer and the layers in the central module. Similarly, the previous context was connected to the context layer with unit weight one-to-one connections. The context layer had fixed all-to-all connections with each layer in the central module. The neurons in the layers of the central module had variable connections to the output layer and the predicted context layer. The spiking neural model used for the central module neurons was the spike accumulation model, similar to the LIF model.

This model is similar to ours, in the sense that it too deals with predictive sequence learning, where the output is a prediction of the next input. However, the learning rule in this model was supervised, based on the minimisation of the error between the current output and the next input. In our model, we use an associative memory and unsupervised learning. Berthouze's model is suitable

for the case where sequences are generated using a specific grammar, and the objective is to efficiently train the model such that it can predict correctly the next symbol from sequences generated from that grammar. Our model, on the other hand, makes no assumptions about the generative model of the sequence, and should perform better in the on-line mode, where there is no separation between the training and the recall stages.

2.8 Representing a sequence: coding theories

In this section, we discuss various theories of how information is coded in neurons.

2.8.1 Rate codes and temporal codes

The brain decodes the enormous volume of spike information that comes from the senses, stores some of it as memories, and transmits commands to the motor and limbic systems also through spikes, all in real time. What is the language used by the brain to encode and decode the spiking information? This problem of neural coding has generated much interest, both among those studying biological mechanisms [65] and those interested in building robots to perform human-like functions.

From the time of Adrian [2] in 1926 until today, various theories have been proposed to explain coding by spikes. Most of them use rate coding, which has been a dominant theory for the past 100 years. Rate coding postulates that information transmitted by neurons is encoded in the rate of firing of spikes. The rate can be expressed in various ways such as averaging over a number of neurons in a specific time window, averaging over a number of runs, averaging for one neuron over a length of time, etc [53].

Temporal coding is another theory which can explain how neurons code information. In temporal coding, the actual timing of the spikes is said to carry information rather than just the firing rate as in rate coding. A number of temporal coding theories have been proposed, such as time to first spike, synchrony between different neurons, and timing of the spike with respect to the phase of the neural oscillation, etc. These can be read in more detail in books by Maass [53] and Arbib [5].

2.8.2 Biological evidence for temporal codes

More recently, Thorpe et al have argued [77] that the timing of the spikes cannot be ignored (as in rate coding) as it is important in processing in the brain. Thorpe showed that in some experiments involving fast object recognition in the visual system from the retina to the layers of the cortex, the spikes from the retina had to pass through a number of layers, and the processing time to recognise images was only of the order of a few hundred milliseconds. This meant that for some layers in the brain, there is only enough time for at most a single spike per neuron to pass, thus making rate coding unlikely. Thorpe argued on this basis that the first few spikes were the most important in encoding and decoding the image in cases of object recognition through layers of the visual system including the retina, and proposed a model of visual processing that encoded information based on the relative temporal order of spikes and giving more importance to earlier spikes.

In our model, we have used the same temporal coding scheme as Thorpe, called rank-order coding [76], to encode our symbol vectors as bursts of spikes in the sequence machine. This code is an abstraction of true temporal coding, because it is not concerned with the exact timing of the spikes but only with their relative order of firing in a layer.

Panzeri et al [60] conducted some experiments on the somatosensory system of a rat, deflecting the whiskers using materials of different levels of coarseness. They found that most of the information transmitted about the identity of the material, based on the firing of the whiskers, could be extracted from the first spike after the stimulation. The second and later spikes also transmitted some information, but in most cases the stimulus could be deduced from the first spike only. This can be considered as additional supporting evidence for Thorpe's hypothesis that the first few spikes are the most important in transmitting information about a new stimulus.

Johansson et al [42] found evidence for coding by the earlier spikes of the tactile signals at the human fingertips. They found that in some cases, the use of the tactile information at the fingertips was faster than could be explained if the information had been transmitted by rate coding alone. They also found that the first spikes transmitted the most information about the stimulus.

2.9 Context in the sequence memory

In building a memory to remember sequences, we need to have some representation of the context of the symbol in the sequence. We have seen previously that many biological as well as artificial neural network models have some way to store the context. This context memory can be stored either as an internal neural state or given an external explicit representation (some separate neurons to represent the context). In our model, the use of context is primarily inspired by the Elman model [23] as well as the presence of some form of context storage in most biological models of memory.

In some models such as Hawkins' model of the neocortex [31] and also Kanerva's sparse distributed memory [45], hierarchical memories are used to store sequences of sequences instead of a separate memory for context. However, we choose to use a separate context memory for our problem rather than a hierarchy of memories, which is a simple solution for our on-line learning problem. Also, in our model we will store the context as a short-term rather than long-term memory, i.e. we will have a neural layer with fixed weights for the context rather than having learning for the context layer too. Learning will be confined to the main associative memory, which will associate the context (history) with the present input.

2.10 The context encoding problem

In chapter 2 we explained why it is necessary to have some representation of the context or past history of the sequence in the sequence machine in order to associate it with the new input. In this section, we examine how to encode the context efficiently in our context memory.

The context encoding problem is to determine the representation or encoding of the context or history of the sequence in the most efficient way possible, so that we can recover the whole sequence in an associative memory, on presenting this context as a cue.

2.10.1 Encoding the context: previous approaches

The problem of how to encode the past history efficiently has been investigated in the past. Plate [62] proposed some ways to encode higher-level associations as

a fixed-length vector. He used circular convolutions to associate items (a convolution of two vectors involves a compression of their outer product, by summing along the trans diagonals of the outer product, and a circular convolution is a special type of convolution that does not lead to an increase in dimensionality of the result), resulting in a vector of fixed length to represent the past context, which he called a reduced representation. Circular correlation was used to decode the convoluted vector into its constituent vectors. The compression of several vectors into a vector of fixed length was possible because a convolution trace only stored information sufficient for distinguishing the constituent vectors, but not enough to reconstruct them accurately by itself, needing input cues to recall the stored data accurately.

For storing sequences, Plate proposed several methods including storing the entire previous sequence as context, using a fixed cue for each position of the sequence, chunking sequences by storing a combination of symbols constituting the sequence, or storing a convoluted version of the previous sequence. For example to store the sequence abc, the convolution trace would be $a+a*b+a*b*c$ (where the * represents convolution) if the entire sequence is used as context, and the retrieval cue would need to be built up also using convolutions.

In our case, since our vectors are coded as ordered N-of-M codes (such as 11-of-256 codes), the result of any convolution of the past input ordered N-of-M vectors to represent the context would have to keep the N same as the constituent codes (a restriction not necessary in Plate's model), thus losing some information.

As discussed earlier in chapter 2, we have two restrictions in choosing an appropriate way to represent the context: one is that it should give priority to nearest inputs to be tolerant to errors. Second, it should be able to disambiguate similar looking sequences and therefore should not forget the past completely. We want a machine that can balance these two considerations.

In the next two sections, we will look at two possible ways of encoding the context, which encapsulates the past history of the sequence. We will examine the models in light of the on-line sequence learning framework that we described in chapter 2.

2.10.2 Context as a shift register

One way to encode the context or the history of the sequence is as a fixed-length sliding time window (or lag vector) of the past, and associate the next output

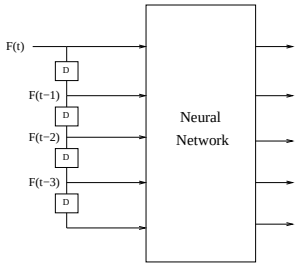

Figure 2.5: Using delay lines to represent time explicitly in a Time delay Neural Network (TDNN)

with inputs in the time window, as is done in Time Delay Neural Nets (TDNN) ([51], [83]), as shown in figure 2.5. The input is stored in some form of memory and fed back with some delay to form the lag vector. Such a memory acts like a shift register, in the sense that the part of the previous input it remembers is 'shifted' with each discrete time interval. Relating this model to the on-line learning framework described earlier, here in Step 2 the new context will be obtained by adding the input to a shifted version of the old context.

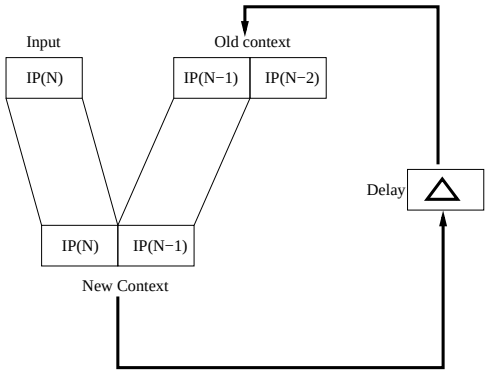

Figure 2.6: Creation of the new context from the old context and the input, using a shift register with a time window of 2. The context at discrete time N is formed by the input IP(N) and the shifted version of the old context, which stores the past input IP(N-1). During the shifting of the old context, the other half of the old context is discarded. The present context vector is fed back to the context layer after the delay of one timestep.

Fig. 2.6 shows the design of a shift register model. Let us suppose the two sequences to be learnt are ABCDE and WXYDZ, and the length of the window is two discrete time units (i.e. the memory stores the past two inputs as the context). The memory will then learn the associations $AB \rightarrow C$, $BC \rightarrow D$, $CD \rightarrow E$ in

the first sequence, and $WX \rightarrow Y$, $XY \rightarrow D$, $YD \rightarrow Z$ in the second. So it can successfully disambiguate the successor of D in the two sequences. An advantage of using a shift register model is that it can retrieve the rest of a stored sequence of any length by giving any inputs from the middle of a sequence, not necessarily from the beginning. Also, we can have hierarchical contexts, by having higher level time windows, as in the approach taken by Kanerva [45]. The disadvantage in using this model is that the time window is of fixed size, and the number of common patterns might be greater or smaller than the size. The shift register forgets the old context beyond the look-back, which is the size of the time window. For example, in storing the sequences ABCDE and UVCDW, if we use a time window of 2 in the shift register, the machine cannot disambiguate the successor of CD.

In the shift register model, let us assume we have a shift register having a time window of 2, which remembers the value of the inputs from the two previous time instances. If the size of the input vector is M, the size of the context vector will be 2M (double the size of the input vector, representing the value of the input at the two previous time instances). In such a model, F is a nonlinear function representing the shifting of the first M bits from the context vector (representing the old input) to the back and M bits from the new input to the front of the new context vector.

The algorithm for the formation of the new context in a shift register model with lookback of 2 is shown in figure 2.7.

2.10.3 Context as a nonlinear encoding

Another approach is to use a separate 'context' neural layer to encode the context, rather than a fixed-length time window. This separate neural layer stores a representation of the entire context or past history of the sequence, rather than just the last few patterns as in a shift register model. A neural layer with fixed weights gives a nonlinear encoding, and thus can be used to produce the context. In such a memory, when we give an input pattern and want the output according to the sequences previously learnt by the memory, it is determined by the present input as well as the output of this context layer. The present output from the context layer is a function of the present input and the fed back previous context representing all past inputs.

Such a model resembles a finite state machine or FSM (see figure 2.8) in

Time is in discrete timesteps t ϵ {1,2,3,...,n,...}

If the following notation is used to represent the input and context vectors:
I_n = input vector at the end of the n^{th} timestep, whose dimension is M.
C_n = context vector at the end of the n^{th} timestep, whose dimension is 2M.

In a shift register with lookback of 2,
$C_n = F(C_{n-1}, I_n)$

where the function $F : [0,1]^{3M} \rightarrow [0,1]^{2M}$ representing the mapping from the input and the context to the new context is defined as :
$[C_n]_k = [I_n]_k$ if $1 <= k <= M$
$[C_n]_k = [C_{n-1}]_{k-M}$, if $M + 1 <= k <= 2M$

The expression for an L-shift register, where L is the lookback, is:
$C_{n+1} = F(I_{n-1}, I_{n-2}, I_{n-3}, ..., I_{n-L+1})$

Figure 2.7: Formation of new context as a function of the previous context and input, in shift register models such as TDNN's [51, 32]

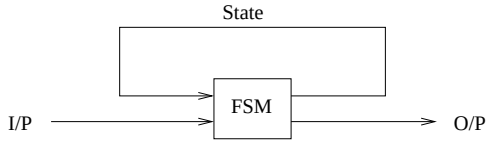

Figure 2.8: Structure of a finite state machine

the sense that the output is a function of the context (representing the internal state of the FSM) and the new input. As described earlier in this chapter, such models were used by Elman [23] and Jordan [43], both of which ran on supervised learning, based on minimising the error in the output when the context was presented.

However, we can use a context neural layer with unsupervised learning as well, for example by having an associative memory that writes the association of the context and the input.

The algorithm for the formation of the new context in the context memory model is shown in figure 2.9.

Time is in discrete timesteps t ϵ {1,2,3,...,n,...}

If the following notation is used to represent the input and context vectors :
I_n = input vector at the end of the n^{th} timestep.
C_n = context vector at the end of the n^{th} timestep.

In the context memory model,
$C_n = \Theta(W I_n + C_{n-1})$
where
W is the weight matrix connecting the input to the context.
Θ is the threshold function of the context layer.

Figure 2.9: Formation of new context as a function of the previous context and input, in the context memory model, similar to that used by Elman [23]

Relating the context neural layer model to the on-line sequence learning framework described in chapter 2, here in Step 2 the new context is the output of the context neural layer, whose inputs are the old context and the present input symbol. The new context will be the output of the context neural layer with fixed weights whose inputs are the fed-back previous context and the input. Thus the context encodes the entire past history or 'state' of the sequence.

Such a model should theoretically give unlimited look-back, since the context is a function of all the past inputs, but practically the activation patterns will decay over a few cycles [34]. However, a problem with this model is that to retrieve a previously stored sequence, we need to start the retrieval from the beginning of the sequence. A learnt sequence cannot be predicted by giving a symbol from the middle of the sequence as cue. This is because the nonlinear encoding of the context neural layer gives equal weighting to all the inputs, not biased towards the more recent inputs as we might desire if we need to retrieve a sequence from the middle. Thus, there is a trade-off between giving precedence to the more recent inputs (thus slowly forgetting the past) and remembering the past context (in order to predict accurately based on the past).

2.11 Conclusion

In this chapter, we have provided an introduction to our problem and reviewed models of sequence learning. In the next chapter, we shall cover spiking neuron models and associative neural memories, including the sparse distributed memory.

Chapter 3

Neural models of learning

This chapter gives a short introduction to spiking neural models and associative memories, including the unordered N-of-M SDM memory that is relevant to our work.

3.1 Artificial neural networks

Artificial neural networks seek to model natural neurons by abstracting some features of their behaviour. There are many models of artificial neurons depending on their connection structure, the algorithm used for learning or changing the connection values etc. Artificial neural network models can be classified in various ways, depending on the architecture, encoding (linear or non-linear) or decoding (closest match or others) schemes, static (does not learn) or adaptive memory, learning rule used (correlation or reward based), closed loop (recurrent) or open, connection weights based or weightless models, etc. Artificial neural networks have many useful applications in fields as varied as weather forecasting, face, handwriting or voice recognition, time series, sequence and structure analysis, and so on.

An useful introduction to common artificial neural network models can be found in the book by Haykin [32]. Some common types of artificial neural networks are self-organising maps, associative memories, multilayer perceptrons using gradient-based learning rules such as backpropagation, radial basis function (RBF) networks etc. Most of these can be implemented by using nonlinear neural models such as sigmoidal neurons, which are based on rate coding (introduced in Chapter 2). However, many of these models can be implemented using spiking

neurons as well. Most neural networks store a nonlinear function of the encoded inputs, and use it to compute the outputs.

An artificial neural network can be used as a classifier (to group data into classes), as a memory (to store and retrieve associations of items of data), as control logic elements, as predictors (building an internal model of a time series and using it to predict subsequent items in the series), etc. In our sequence learning model, we use an artificial neural network as an associative memory implemented in spiking neurons and use it to predict the next terms in the sequence. We will concentrate on associative memories in the following section.

3.2 Modelling of spiking neurons

Various spiking neuron models have been used to abstract the essential elements of the spiking behaviour. These vary in the level of abstraction and the variety of spiking phenomena modelled, ranging from very simple models (such as the standard leaky integrate and fire neural model [53]) to very complex compartmental models (which model the behaviour of various regions of the neuron as compartments with the help of cable theory (originally studied in 1850s by Lord Kelvin for application to transatlantic cables and later by Rall [63]) and differential equations to model the dynamics of ion channels, such as the Hodgkin Huxley model). A popular generic spiking neuron model is Gerstner's spike response model [28].

The dynamical systems approach is to treat the spiking neuron as a dynamical system and have a set of equations to model the most essential behaviour without worrying about the accuracy of the biological structure in the modelling, as in the pulse-coupled models of Wilson [85]. Another of these models is Izhikevich's model [36] which can simulate a wide variety of spiking phenomena such as bursting.

More information about spiking neural models of differing complexity can be found in books by Koch and Segev [48], Maass [53], etc. Books by Wilson [85] and Feng [25] give an overview of the pulse-coupled and other dynamical models.

3.2.1 Choosing a spiking neural model: assumptions and tradeoffs

In our modelling of spiking neurons, we have made a few simplifying assumptions. We are not concerned with the shape of the neuron and treat them as point neurons. We assume that all spikes look alike (in the sense that the activation increases until the threshold is exceeded and then it is reset) and so the shape of the spike does not matter. We also assume a spike generation mechanism, whose low-level details we are not concerned with.

The choice of spiking model to use usually depends on the application. Simple models abstract all but the most essential characteristics of real neurons and are fast to compute. However, it is debatable which characteristics are essential to the computation being performed by the neuron, so there is the danger of leaving out essential behaviour in oversimplification when using the simple models. On the other hand, more complex models such as the compartmental models are more biologically accurate and capture most of the known spiking behaviours, but are very slow to run and are complicated to model with. Genesis [14] and NEURON [15] are common spiking neuron simulators, that can model a variety of low-level behaviours like the dynamics of different ion channels.

For our purposes, all we require is a simple model having few parameters that is easy to work with. The leaky integrate and fire model is a candidate to model our neural behaviour. It is discussed in the next subsection.

3.2.2 Leaky integrate and fire model

In the leaky integrate and fire (LIF) model [53], the activity of a neuron can be described as an analogy with an electrical circuit, consisting of a capacitor C in parallel with a resistor R driven by a time varying driving current I(t), as illustrated in figure 3.1. The membrane potential or activation of the neuron is represented by the quantity u(t), which is the potential difference across the capacitor or resistor. The current I(t) splits between the capacitor and the resistor.

$$I(t) = \frac{u(t)}{R} + C\frac{du}{dt} \tag{3.1}$$

Multiplying the above equation by R, if we call the quantity RC as the driving constant τ (representing the membrane time constant of the neuron), we get:

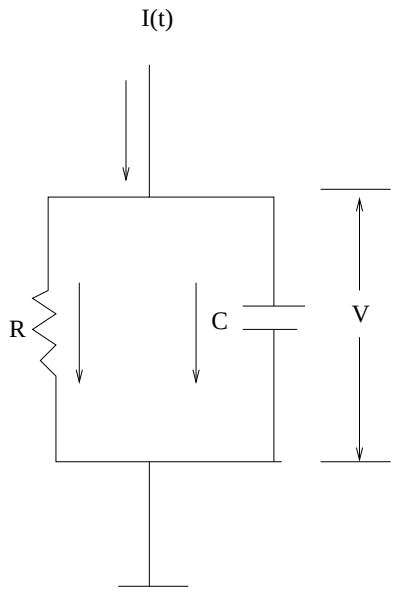

Figure 3.1: Leaky integrate and fire (LIF) model of a neuron, represented as an electrical circuit

$$\tau \frac{du}{dt} = -u(t) + RI(t) \tag{3.2}$$

If we take the resetting membrane potential as 0, the input driving current I(t) as a constant value I, we get the expression for the activation of potential at time t as a result of an incoming spike at time t_0

$$u(t) = RI[1 - exp(-\frac{t - t_0}{\tau})] \tag{3.3}$$

Incoming spikes to a neuron increase its activation, which decays back to the resting potential with time in the LIF model. Thus, the model 'integrates' input spikes.

When the membrane potential u(t) exceeds a threshold Θ, the neuron is said to have fired and the membrane potential u(t) is reset to a low value called the resting potential. Let us say the neuron fires at time T.

$$u(T) = \Theta \tag{3.4}$$

Solving the equations we get the value for the firing time T, when the membrane potential equals threshold Θ as follows:

$$T = \tau ln \frac{RI}{RI - v} \tag{3.5}$$

However, we need a neuron model with more complex dynamics than the leaky integrate and fire model for our work. We will examine two different neural models for suitability in chapter 6: the first being a variant of the leaky integrate and fire model with nonlinear dynamics called rate driven leaky integrate and fire (RDLIF) model, and the second being a simpler model with linear dynamics, called the Wheel or firefly model.

3.3 Associative memories

A neural network can be used to store information, and can thus be considered as a memory. An associative memory is a system that stores and retrieves associations of suitably encoded symbols or patterns. It functions in two stages: it learns or writes an association of two encoded symbols, say 'A' and 'B' to the memory during the writing stage, and can read or retrieve a the other half 'B' of the previously written association on presentation of the first half 'A' during the reading stage. A computer memory, where data are stored at specific hardware addresses, can be considered as associating the data with the addresses, and so is also a type of associative memory. Associative memories can be implemented in neurons, and neural associative memories have some added features compared to conventional computer memories. From this point on, we shall use the term 'associative memory' to refer to neural associative memories only, unless specified otherwise.

Associative memories have a finite capacity for the number of associations that can be written, beyond which they can no longer retrieve previously written associations accurately, however the forgetting is gradual rather than abrupt. These memories are also tolerant to input noise to a certain extent. The storage of associations of symbols in these memories is done in a distributed way, with each item of data being stored and retrieved from multiple neural locations.

Associative memories can be used in two ways: auto-associative and hetero-associative. If used as **auto-associative memories**, they associate patterns to themselves, and can improve the quality of noisy patterns. For example, such a memory can associate a pattern 'A' with itself, and on being given a corrupted version of 'A' as input, it can output a version of 'A' that is closer

to the original 'A' than the corrupted version. The Hopfield model [35] is an example of an auto-associative memory having bipolar weights (+1 and -1). If used as **hetero-associative memories**, the two items to be associated do not need to be the same, either in content or in dimension, and one of them can be used as a cue to retrieve the other. Correlation matrix memories [49, 84] are examples of such memories. Associative memories can be unidirectional, with only forward connections from one neural layer to another, or bidirectional, with connections both ways between neural layers, with equal connection strengths in both directions. These memories have also been described as content addressable memories or CAMs, because the address where the data is stored depends on the content of the data to be stored.

A good introduction to associative memories can be found in books by Beale [10] or Haykin [32]. For interested readers there is a detailed theory of associative memories (with respect to their memory capacity and other properties) in papers by Palm et al [66].

In the following subsections, we shall discuss some important associative memory models on which our model is based, namely the correlation matrix memory, the ADAM system with N-tupling, the sparse distributed memory and its implementation using N-of-M codes.

3.3.1 Correlation matrix memories

Correlation matrix memories (CMMs) are a type of hetero-associative memory that store the correlations between pattern vectors to be associated in the connection weight matrix and superimpose multiple correlations. These memories were proposed by David Willshaw [84] in 1969 and a linear version of the memory by Kohonen [49] in 1972. These memories are based on a localised Hebbian learning rule, and learn and recall in a single pass. In these models, neurons have binary states 1 and 0.

Willshaw memories [84] are a type of CMM having binary weights. CMMs are popular due to their good information capacity, that has been shown to be higher than that of the standard Hopfield model [7, 39].

Let W be the connection weight matrix of the CMM and the n^{th} pair of encoded vectors of the symbols whose association is to be written to the memory.

In **writing the association $X_n \rightarrow Y_n$ to the CMM**, the weight matrix W_n

will be given by:

$$W_n = W_{n-1} \oplus X_n Y_n^T \tag{3.6}$$

where Y_n^T refers to the transpose of the vector Y_n, W_{n-1} refers to the state of the weight matrix after $n-1$ associations have been written and $\oplus$ refers to the logical OR operation (in case of a binary CMM) or addition (in case of a linear CMM). If X_n is a vector of dimension $A \times 1$ and Y_n a vector of dimension $B \times 1$, the dimensions of the weight matrix W would be $A \times B$.

In **reading from the CMM**, the first half of the stored association X_n is presented as a cue to retrieve the other half Y_n.

$$Y_n = \Theta(W^T X_n, T) \tag{3.7}$$

where W^T refers to the transpose of the weight matrix W, Θ refers to the thresholding operation if the product exceeds a threshold T. For a binary CMM, Θ will be equivalent to the Heaviside function, setting the vector component to 1 if it exceeds T and 0 if it does not.

3.3.2 Associative memories using spiking neurons

Implementations of associative memories using spiking neurons have also been proposed [47, 72, 50]. In this subsection, we consider some of these models.

Knoblauch's model[47] used a spiking associative memory constructed using three populations of neurons, one excitatory (the actual associative memory) and two inhibitory. An addressing group of neurons gave inputs to the excitory and one of the inhibitory populations, while the other inhibitory population received input from within the excitatory population. It used the spike counter model of spiking neuron, and implemented a Willshaw associative memory [84].

Kustrin et al developed a model of spiking correlation matrix memory [50], based on a direct implementation of the CMM recall equation using the spike response model [28] of spiking neurons. The model had two neural layers, a summing layer, which was fully connected to all the inputs, and a thresholding layer which was fed output spikes from the summing layer and also connected to an auxilliary threshold neuron that was fed input spikes and fired on receiving a certain level of input activity. Willshaw thresholding was performed by using coincidence detection by the neurons of the thresholding layer, the threshold being set to the number of synchronised input spikes.

3.3.3 N-tupling

N-tupling refers to selecting the N highest outputs from a layer of neurons. The Advanced Distributed Associative Memory (ADAM) system, developed by Austin et al [6] was a CMM based associative memory with N-tupling. It used a binary N-tuple preprocessor and two binary CMM layers, the first of which associated the input pattern, pre-processed by an N-tuple decoder, with a class pattern chosen from a list of available mutually orthogonal class codes, and the second associated the chosen class pattern with the desired output. The mappings of both the CMMs were many to one, i.e many inputs could be associated to a single class code, and many class codes could be associated to a single output. During recall, the output class pattern was determined using L-max thresholding, meaning that exactly L highest summed components were set to 1 and the rest to 0. The output of the second CMM was determined using Willshaw encoding. The N-tuple decoder functioned as an application-specific lookup table (no associations were written in this layer) to translate the first half of the association into an N-tuple. The system was used to efficiently store and retrieve images.

The N-of-M SDM associative memory (on which our model is based) used a binary N-tuple address decoder as in the ADAM system, and applied an L-max thresholding to select the maximum components and set them to 1, but had only one CMM layer instead of 2, and used random connection weights instead of a designed lookup table to pre-process images.

3.3.4 Sparse distributed memory

The sparse distributed memory (SDM) is a type of associative memory that has the features of distributed storage, scalability and error tolerance. Since the associative memory in our sequence machine is an implementation of the SDM, an understanding of how the SDM works would help with the understanding of our system as well. The original model was developed by Pentti Kanerva [45] in his book of the same name.

The SDM is like a conventional random access memory (RAM) in that it stores address-data associations. RAMs store a piece of data at a particular address, and the data can be retrieved on giving the address as a cue. Similarly the SDM also associates a ‘data’ pattern to an ‘address’ pattern, and given an address as cue the corresponding data can be retrieved. It differs from a RAM in that even

if the address given was noisy or partially incorrect, the data retrieved by the memory is more similar to the original data than the noisy address was to the original one. On being given a random 'address' pattern, the memory retrieved the data pattern closest in distance to that address.

The SDM can be seen as a two-layer feed-forward neural network. The first layer, called address decoder, has fixed connection weights. The second layer, data store, has changeable weights as a result of learning. While retrieving stored data from conventional RAMs, a given binary address is first decoded and the data is then read out from the hard location corresponding to that address. The two neural layers of the SDM work with similar principles.

Address and data patterns are encoded as high dimensional binary numbers. Now, a RAM having a long address would require an exponential number of locations to store all the possible addresses in the address space. However in the SDM the number of required address locations (which Kanerva called hard locations, referring to hardware needed for the storage) is sparse compared to the theoretical maximum. This is possible since storage is distributed: one item of data is stored at many hard locations, and each hard location stores more than one item of data. Each of these hard locations has an array of up-down counters, one for each bit of the data. While retrieving a piece of stored data corresponding to a given address, all the hard locations within a given Hamming distance from the address are considered.

The principle behind the SDM is that if we think of the address space as a high dimensional binary hypercube where every binary address is a point on the hypercube, most of the points would be at the edge of the hypercube and very distant from each other (average of n/2 distance where n is the dimensionality of the hypercube, [45]). Any two given addresses in the address space would be nearly orthogonal. For any two address points that are distant, typically one can find a point that is close (their Hamming distance is below a threshold) to both. Therefore we can have only a limited number of hard locations and store and retrieve data, in a distributed way, from all the hard locations that are close to the address of the location to be read.

Let the address length in bits (input dimension) be N and data length (output dimension) be U. There are a fixed number M of hard locations that is much less than the size of the address space 2^N. Each of the hard locations will then have U up-down counters, each counter being initially set to 0.

The method for **writing to the memory** is as follows:

When an N-bit binary address A is presented to the address decoder, it activates all the hard locations M_i which are in the access circle of A, i.e. whose address is within a specified Hamming distance Θ (number of bits in which they differ) from A (i.e $d(A, M_i) = A.M_i \leq \Theta$, where the '.' represents the inner or dot product of the two binary vectors A and M_i), and writes the U bit data word into each of those locations. It does this by incrementing or decrementing the U counters connected to those hard locations as follows: if the k^{th} bit of binary input data D_k to be written at the address A is 1, the k^{th} counter of each activated hard location is incremented by 1. If the k^{th} bit is 0, the k^{th} counter is decremented by 1.

While **reading from the memory**, a similar procedure is followed. The N-bit address cue A is presented to the memory as before, and also activates all the hard locations M_i that are within the access circle of A ($d(A, M_i) \leq \Theta$). The counters corresponding to these activated hard locations are summed and thresholded to retrieve the binary output data. For example, if the sum of the k^{th} counters of all the activated hard locations is S, and S exceeds a threshold T, then the k^{th} output bit is set to 1, else it is set to 0.

3.3.5 SDM using N-of-M codes

In this subsection, we describe an implementation of the SDM using an encoding called N-of-M codes. This implementation was originally proposed by Furber et al [27]. We have used a modified form of this model in the associative memory part of our sequence machine, which we shall describe in chapter 4.

An N-of-M code has a total of M components, out of which exactly N are active at any given time to form a valid symbol, and the choice of the N determines the code. Such a code is self timing or self error-detecting, as codewords with errors in the number of active components can be detected. Sparse codes are used for this memory, with $N << M$, similar to Kanerva's original model. Sparse codes have been shown to have high information capacity [66]. The N-of-M coding was inspired by Austin [7] who used the N-of-M codes (or N-tupling) in the ADAM system as described earlier.

The N-of-M SDM model combines the standard SDM with a correlation matrix memory layer. It consists of two layers of neurons as in the original Kanerva SDM: an address decoder layer and a data store layer. Both the address and

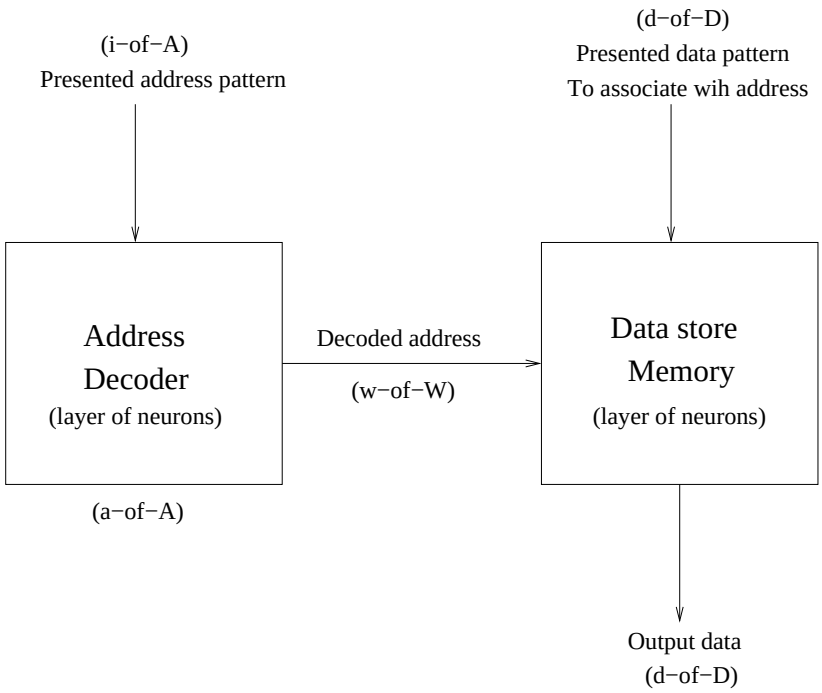

Figure 3.2: The N-of-M neural memory model

the data pattern vectors are encoded as N-of-M codes. The number of address decoder neurons is much greater than the number of bits in the address, and the primary purpose of the address decoder layer is to cast the input address symbol into a high dimensional space to make it more linearly separable. The data store layer in the N-of-M SDM is different from Kanerva's second layer. In Kanerva's model, the data store layer consisted of a number of up-down counters. In Furber's model, the data store is a correlation-matrix memory which associates the first symbol as decoded by the address decoder layer with the second symbol. Learning takes place only in the data store layer, while the weights of the first address decoder layer stay constant. Furber's paper also demonstrated that such memories were scalable and error tolerant [27].

Working of the N-of-M SDM

Let us assume that the memory consists of W address decoders and the input data is a d-of-D coded pattern. There are D neurons in the data memory. To cast the address into high dimensional space, the number of address decoders W is chosen to be much larger than A, the dimensionality of the address pattern. The input address is an i-of-A code. Here i may or may not be equal to d and A

to D.

Write operation

In writing an association $x \rightarrow y$ to the memory, an i-of-A coded address word x is associated with a d-of-D coded data word y. The address word x is input to the address decoder layer, whose weights are set to a random a-of-A code. A threshold t_A on the activations of the address decoder neurons is chosen such that approximately w out of the total of W address decoders fire, or selecting w neurons with the highest activation, and in case of ties selecting all the neurons whose activations are equal to the activation of the w^{th} neuron.

$$w = \Theta(Ax, t_A) \tag{3.8}$$

where Θ is the heaviside function, setting the vector component to 1 if it exceeds threshold t_A and 0 if it does not.

Initially the data memory, which is the second layer in the model, is empty, so its weights are set to 0. The d-of-D coded input data is presented to the D data memory neurons. Writing to the memory takes place by the following rule: if the k^{th} data memory neuron is 'active', meaning that if the k^{th} bit of the data input has been set, and the i^{th} address decoder neuron fired, the i^{th} weight of the k^{th} neuron is set to 1, if it was not set earlier. Thus it uses the OR function to set the weights (the new connection weight is the OR of the corresponding input bit and the address decoder bit), as in binary CMMs.

$$D = D \oplus wy^T \tag{3.9}$$

where $\oplus$ indicates logical OR, and y^T denotes the transpose of the row vector y.

Read operation

First the i-of-A input address word x is presented to the W address decoder neurons, resulting in w of them firing.

$$w = \Theta(Ax, t_A) \tag{3.10}$$

where w is the binary high dimensional word line vector that is output by the address decoder, A is the address decoder binary matrix of dimensionality $W \times M$, Θ is the threshold heaviside function which sets each element of the matrix to 1 if it exceeds threshold t_A and to 0 otherwise.

The data memory output y is computed by taking the dot product of the address decoder firing array with the data memory weight matrix. Since the output data has to be a d-of-D code as well, the resulting activations are sorted and the d maximum activations chosen.

$$w = L(D^T w, d) \tag{3.11}$$

where L is the l-max function, setting the largest d elements to 1 and all others to 0.

The N-of-M modification to Kanerva's SDM has been shown to be more information-efficient [27] than the original SDM.

3.4 Conclusion

In this chapter, we reviewed the LIF model of spiking neuron and different kinds of associative memories, introducing an N-of-M SDM memory. In the next chapter we shall introduce a novel associative memory using the N-of-M SDM and rank ordered codes, which we shall use in the construction of the sequence machine.

Chapter 4

A Novel rank ordered N-of-M SDM memory

In this chapter the associative memory model used in the sequence machine is described. Our chosen model is a sparse distributed memory (SDM) using rank-ordered N-of-M codes. It is a further modification to the unordered N-of-M implementation [27] of Kanerva's SDM [45] which we described in chapter 3. Our modification to Furber's model is in using rank-ordered [77] N-of-M codes and real-valued weights for the memory instead of unordered N-of-M codes and binary weights. This modification facilitates the implementation of the memory using spiking neurons.

We examine factors that effect the performance of the memory, and examine the suitability of rank-order codes in the memory as compared to unordered codes. First of all, we shall describe the coding scheme itself, and how to represent it.

4.1 Rank-order codes

Rank-order codes were first used by Thorpe [76]. The concept of rank-order is simple: the relative temporal order of firing of neurons in a layer makes the code. As an illustration, if a neural layer has 5 neurons numbered 1 to 5, an example of a valid code representing an output burst of spikes fired by the layer is $[4, 2, 1, 5, 3]$, which stands for neuron 4 firing first in the burst, followed by neuron 2, followed by 1 etc. Rank-order coding is a simpler or abstracted version of temporal coding, in which we consider only the order of firing of the neurons in a layer rather than their actual times of firing. However, rank-order codes do have more temporal

information than rate codes, since they take into account the temporal order of firing of neurons in the layer.

In chapter 3, we mentioned that Thorpe et al postulated that the earlier spikes were the most important in recognition of an image, because in certain cases the brain has time to fire at most a single spike through each layer, making rate coding infeasible [77]. Based on that they described a decoding scheme for making a neuron sensitive to a particular input order of spikes, by giving more importance to earlier input spikes and progressively less importance (by desensitising the layer) to later input spikes from a layer, engineering the input weights and modulating the threshold of the neuron to be receptive to the specific input order [76]. They used such a scheme in a biologically-plausible model of the visual system for efficient face recognition from a huge database of face images.

4.1.1 Rank-ordered N-of-M codes

In chapter 3 we described Kanerva's Sparse Distributed Memory and an implementation of it based on N-of-M codes, as used by Furber et al [27]. N-of-M codes are unordered codes of M bits, out of which only N are active ($N << M$). For example, an unordered 11-of-256 code will be represented using a binary vector of 256 bits, out of which only 11 are 1 and the rest are 0 in a valid code. Rank-ordered N-of-M codes are similar to unordered N-of-M codes, with the rank of the selected bits taken into account as well. An example of a 3-of-5 ordered code representing 3 neurons firing in order in a layer of 5 neurons (numbered from 1 to 5) is [2,4,1].

Rank-order codes have good information content, based on the number of possible permutations from such a code. The number of possible ordered N-of-M codes is the number of ways one can choose N items in order from a total of M items, which is $\frac{M!}{(M-N)!}$. The information content of a word, measured in bits, would be log to the base 2 of this value. The information content of an unordered N-of-M code is log to the base 2 of the number of ways one can choose N items out of M without caring for their order, which is $\binom{M}{N}$ or C_N^M or $\frac{M!}{N!(M-N)!}$. This is less than the information content of the corresponding ordered code, as illustrated in figure 4.1.

Our reason for our choosing to use rank-order coding in addition to N-of-M coding, apart from the increased information content, is that we intend to implement the memory using spiking neurons. Spikes fire in real time, therefore

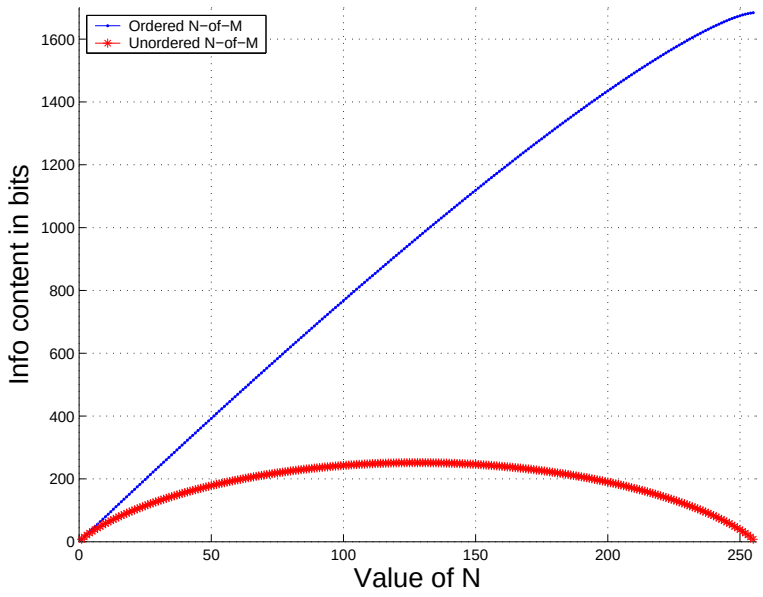

Figure 4.1: Plot of the information content in bits of an unordered (red) and ordered (blue) N-of-256 code, when N is varied from 1 to 255. The unordered memory curve peaks at the half way point of N=128, where the value of the information content is 251.673 bits. The ordered memory curve rises constantly, reaching 1684 bits at N=255.

encoding a vector representing firing times of a group of spikes as a binary code, which would lose the timing information, is not appropriate. Besides, real-valued connection weights can better store the graded effect of the rank-order sensitivities of neurons than binary weights.

4.1.2 Representation of the code using significance vectors

In the implementation of the sequence machine, we shall use rank-ordered N-of-M codes to encode the symbols being associated in the memory. We need a way to represent an ordered N-of-M encoded symbol representing an order of firings in a burst emitted by a layer of spiking neurons, in order to store the symbols as vectors in the memory. This would be useful in converting a temporal code into a spatial code by abstracting out time from the burst of spikes. Using this method, we can perform calculations as in conventional neural networks (not using spiking neurons), where we multiply the input vector with the connection weight matrix of the neural layer to obtain the activation or output vector.

We encode a rank-ordered N-of-M codeword representing an output order of firings such that the effect of earlier spikes is more significant than later ones in increasing the activation of the target neuron, as per the method first described by Delorme et al [76]. We do this by having a quantity called **significance** for each component of the encoded vector, and using a monotonically decreasing function of the order of neural firings to calculate the significance of each component. The significance represents the sensitivity or effect of incoming spikes in increasing the activation of the target neuron. There can be a number of choices of this significance function. Using an arithmetic progression to encode a symbol as a vector of significances, the first neuron to fire is given a significance of 1, the second one a significance of $1-x$, where $x < 1$, the third neuron to fire $1-2x$, and so on. In a geometric progression the significances would be in a decreasing geometric ratio such as 1, α, α^2 and so on, where $\alpha < 1$ is the significance ratio (or desensitisation factor, as it is the factor by which the layer gets desensitised to successive input spikes). One advantage of using a geometric progression to encode vectors in the system is that the component values cannot go below zero (and we would like to keep the effect of inputs positive, unless the connection weights themselves are negative), as is possible with the arithmetic progression choice. Therefore, for the purpose of our work, we choose to encode the significances via geometric progression.

As an example to illustrate the geometric significance ratio, if we are using an ordered 3-of-5 code in a layer of 5 neurons numbered 1 to 5, we can represent neurons 3,2,4 firing in order as the vector [0,α,1,α^2,0]. We intend the effect of this time abstraction to be the same as a spiking neural system. In other words, we want the output order of spikes of a layer when it gets a particular input order (in a real spiking system) to be equivalent to the output vector calculated on multiplying the time-abstracted input vector (as determined in the way mentioned) with the connection weight matrix of that layer.

Together with this encoding scheme, we also need a way of comparing any two ordered N-of-M codewords, as a metric to determine the 'similarity' or 'closeness' of the two codes. This is needed to decode or identify one of the symbols of our alphabet as being closest to the encoded vector. We choose the dot product of the significance vectors as a measure of similarity. Hence, the similarity between two ordered N-of-M codes X and Y is defined as having the value X.Y, where $X.Y = \Sigma_{i=1}^{n} X_i Y_i$ refers to the dot product of the components. The reason we

chose the dot product to measure similarity is because it is similar to the way the activation is calculated: the activation of a neuron or a neural layer is usually calculated as the dot product $A = W.X$ of the input vector X and weight vector W (taking the sum of the products of the corresponding components). We normalise the dot product to standardise the similarity value. Thus, the expression for the normalised dot product will be as follows:

$$\bar{X}\bar{Y} = \frac{\Sigma_{i=1}^{n} X_i Y_i}{\Sigma_{i=1}^{n} X_i^2 \Sigma_{i=1}^{n} Y_i^2} \tag{4.1}$$

The normalised dot product of two vectors is equal to the cosine of angle between the vectors, or the degree of overlap. If two vectors are identical, their normalised dot product is 1 and so the similarity metric calculated in this way will also be 1. If the two vectors are not identical, their calculated similarity metric will be less than 1 (and proportional to the degree of their overlap), which is what we would expect.

The normalised dot product measure of similarity is related to the commonly used Hamming distance as well. The Hamming distance $H(x, x')$ between two binary codewords x and x' is the number of bits where they differ, and the dot product of binary codewords $x.x'$ gives the number of corresponding bits they are similar. So there is a linear relation between both schemes in the case of unordered (binary) codes, which is as follows:

$$x.x' = 1 - \frac{H(x; x')}{2N} \tag{4.2}$$

4.1.3 Some notes on using the significance vector and normalised dot product

Using a significance ratio of 1.0 is equivalent to having an unordered memory. Any value of the ratio less than 1.0 can distinguish the ordering information.

Using a small value for the significance ratio α in the significance vector would decrease the importance given to later bits, since the significance would decrease faster and have small values for later bits, and the similarity of two vectors would be heavily influenced by the first few bits. Therefore, in our memories, we have typically used the value of α between 0.9 and 1, as we seek to maximise the information content of the memory.

In our sequence memory model, we have a fixed alphabet from which symbols

are chosen, and the number of permissible ordered N-of-M codes is much greater than the size of the alphabet. This allows us some redundancy in identifying which symbol the output is closest to, as we can select the symbol which is closest to the output if the output does not match any of the symbols exactly.

4.2 N-of-M SDM using rank-ordered codes

In chapter 3 we described the operation of an SDM using unordered N-of-M codes [27]. In the implementation of the sequence machine, we are using rank-ordered codes. The reason we chose to use rank-ordered codes instead of unordered codes is the higher information capacity of rank-ordered codes, as shown in section 4.1.1. Also, rank ordering is suitable for implementation using spiking neurons, as shown in papers by Thorpe et al [76, 21].

For implementing an SDM using rank ordered N-of-M codes, we use a similar memory as in the unordered model, but make the following two changes:

1. The symbols are encoded as ordered N-of-M codes rather than unordered codes.

2. The connection weights in the memory have real values rather than binary values as in the original N-of-M SDM. This is to enable the memory to store rank-ordered codes and also to facilitate implementation by spiking neurons.

The ordered N-of-M SDM has two layers of neurons as in the original unordered model: an address decoder layer of high dimensionality (whose weights are fixed) and the data store which is a correlation matrix memory (CMM) but with real weights, which are set during learning. There is one difference between the CMM used in the rank-ordered memory and the one used in the original unordered memory: the unordered memory used the OR function to set the weights, with the new weight component W_{ij} being the OR of the respective components of the vectors being associated, X_i and Y_j. For real-valued weights, however, the OR function is replaced by the MAX function, with the new weight component the maximum of the old weight and the product of the components of the vectors to be associated. The MAX function we use in this case has the same effect as the OR function when used for binary weights (since the maximum of two binary

numbers is the same as the OR of both), so the unordered and ordered memories are functionally similar.

Let σ_i = Significance of the i^{th} bit of the address decoder word line

σ_j = Significance of the j^{th} bit of the data word line

The weight matrix component w_{ij} after writing the association

$w_{ij} = max(w_{ij}, \sigma_i \sigma_j)$

where $max(i, j) = i$ if $i >= j$ and j if $i < j$

Figure 4.2: The learning rule used in the data store layer of the ordered N-of-M SDM

The learning rule for updating the weights of the data store layer is shown in figure 4.2.

The operation of the SDM can be described as follows: Suppose we have to associate symbol 'B' with the symbol 'A', such that if during memory read we give 'A' as input, we should get 'B' as output. The symbols are encoded as ordered N-of-M codes using significance vectors. The address decoder first gets the ordered d-of-D vector corresponding to the encoded symbol 'A', and the data store gets the ordered d-of-D vector corresponding to the symbol 'B'. The address decoder layer (having in total W address decoder neurons) has fixed weights (each address decoder being connected to d out of D input neurons, with the weight matrix also following ordered d-of-D code), and outputs a much longer word (following an ordered w-of-W code) which is fed as input to the data store. The data store writes the association between the address decoder output and the input data using the MAX function as described above. The process of reading from the memory is similar to writing. We first give the encoded ordered d-of-D vector corresponding to the symbol 'A' to the address decoder, whose ordered w-of-W output is fed to the data store as during the write operation. Since the data store has already written the association $A \rightarrow B$, it outputs the encoded ordered d-of-D vector (if we sort and order d neurons with the maximum activations from the output layer) corresponding to the symbol 'B' when it receives the address

decoder outputs.

4.2.1 Tradeoffs of using rank-ordering

Using rank-ordered codes N-of-M in the SDM has the following advantages over unordered (binary) N-of-M codes :

1. Higher information capacity: the information content of a rank ordered N-of-M code is higher than that of a comparable unordered code, as shown in Section 4.1.1.

2. More flexibility to the memory: Using rank ordered codes gives a larger memory space than unordered codes. The number of possible ordered N-of-M codes is P_M^N, which is higher than the C_M^N number of unordered N-of-M codes.

The disadvantage of using rank ordered codes as opposed to unordered codes is the greater hardware cost of using real-valued weights instead of binary weights. Binary weights need only a single bit of hardware for each memory location, while rank order codes would need 8 or 16 bits or higher, depending on the desired precision.

4.3 Performance of the memory

In this section we attempt to measure the performance of the sparse distributed memory using rank-ordered N-of-M codes.

The unordered N-of-M SDM using binary N-of-M codes [27] has been shown to have the properties of good information density, scalability and error tolerance, both analytically as well as empirically through experiments. As described earlier, the SDM using ordered N-of-M codes is similar to the unordered N-of-M SDM and is therefore expected to have similar features. The experiments we perform will be to verify these properties of the memory.

In the context of the implementation of the sequence machine, performing tests on the ordered N-of-M SDM memory is relevant in order to study the factors which effect the memory performance of the sequence machine as well, since it forms the associative memory component of the sequence machine.

4.3.1 Parameters to measure performance

To measure the memory performance, we write a number of address-data pairs to the memory and then try to recall the data vectors by giving the address vectors as cues, each time measuring how accurate is the recovery (measured by a number of parameters), how many address-data pairs can be stored and recalled correctly before the memory saturates and also how the performance degrades after this point. There are two parameters we can use to measure the performance in this way:

- **Perfect match capacity (or absolute capacity)**: This measures the maximum number of stored symbols that can be retrieved to a specified accuracy (a symbol is said to be retrieved correctly if the similarity of the retrieved symbol and what was originally stored in the memory is greater than a specified threshold) before the memory saturates (i.e. the number of words correctly retrieved does not increase as more words are written to the memory, but rather starts decreasing). An interesting experiment is to determine how this measure scales with increasing memory size.

- **Average recovery capacity (or average quality or average similarity)**: This measures the average accuracy of recall of the stored words (measured by finding the similarity between the retrieved word and what was originally stored in the memory, and taking the average of these similarities over all the address-data pairs that were written to the memory). We can also determine how the average similarity changes with increased number of words put in the memory of a given size, and how it scales with increasing memory size.

- **Information density**: This is measured by the amount of information (Shannon information) stored per unit memory (weight matrix) size, in "bits per synapse", a synapse standing for a connection between two neurons or an element of the connection weight matrix. The amount of information in each codeword can measured by the Shannon information measure, which is log to the base 2 of the total number of codewords possible.

$$I_w = log_2 W \tag{4.3}$$

where

I_w=Information content of codeword W

W=total number of codewords that can be formed from the alphabet.

Thus, we choose to define the bit-per-synapse information density of the memory as the maximum perfect match capacity (which is the peak of the absolute capacity curve defined above) times the information content of a codeword, divided by the memory size.

In our experiments, we will plot the memory capacity with respect to the first two parameters of perfect match capacity and average quality.

4.3.2 Memory properties that can be measured

There are a few properties of the memory we can measure with the help of the above mentioned parameters:

- **Scalability**: This is determined by how well the performance of the memory (perfect match capacity or the average quality) scales as the dimensions of the memory are increased. In the ordered N-of-M SDM, the dimensions are d, D (from the d-of-D code used to encode the address and data pairs and also the size of the outputs), w and W (from the w-of-W code used for the outputs of the address decoders). In our experiments we will plot the memory performance with varying memory sizes on a graph and then determine if the memory can be considered to be scalable.

- **Effect of ordering**: This is determined by how the memory performance (either as the perfect match capacity or the average quality) depends on the choice of significance ratio. In the experiments, we vary the significance ratio α giving it values ranging between 0 and 1.0 (α 1.0 represents an unordered code, and other values of alpha represent ordered codes) and plot the memory performance of the ordered N-of-M SDM, keeping all the other SDM parameters (d, D, w, W) constant.

- **Error tolerance**: This can be measured either by increasing the number of bit errors in the address input to a memory of given size, or by putting one bit error in different positions and measuring how well the memory performs with respect to the two parameters mentioned earlier. An alternative method to determine the error tolerance could be to calculate the average

dot product of the significance vectors corresponding to the erroneous words input to the memory and the words without error, and compare it to the average dot product on the output, in order to determine if the memory can improve the average quality of the input words. Such a measure has been used in another paper [26] on N-of-M SDMs (that used a binary memory to store ordered codes). However, in this dissertation we shall stick to the first method of varying bit errors.

Therefore, in the experiments, we have varied the following three parameters: (each time plotting both absolute memory capacity up to similarity 0.9, and average dot product)

1. **The memory size**: to determine if the memory is scalable.
2. **The significance ratio (or desensitisation factor)**: to compare ordered vs. unordered codes and see how the choice of this ratio effects the memory performance.
3. **The number of errors** in the composition of the address vectors given to the memory when the memory is being read out: to determine the error tolerance of the memory.

4.3.3 Experimental procedure for the performance tests

We have performed a number of numerical experiments to measure the memory capacity according to various parameters. They are performed using the following steps:

1. We **initialise the network parameters** (dimensions of the address or data vectors such as 11-of-256, dimensions of the address decoder vector such as 16-of-4096, value of significance ratio α etc). We then generate the weight matrix of the address decoder layer based on an ordered d-of-D code for the input weight array of each of the W address decoders. In other words, each address decoder has non-zero weights for exactly d out of the D input neurons it is connected to, and these non-zero weights are randomly set as per the [1, α, α^2, ..] significance vector. Exactly d random values between 0 and 1 are generated from a uniform distribution and then sorted to find a random order for the input weights in each case. The weight matrix of the data store memory is initialised to 0, since it is initially empty.

2. We **generate n random address-data pairs as encoded significance vectors**. The n address-data pairs are generated according to the d-of-D code with the same α as significance ratio (The value of α can be different from the ratio used to initialise the input weights of the address decoder layer, but for the sake of simplicity we have kept them the same).

3. **Writing step**: We write each of the n address-data pairs to the memory as per the following procedure: first we pass the address vector through the address decoder and the address decoder output vector through the data store layer, where it is associated with the data vector using the MAX function as the learning rule. At the end of this step the memory is filled with n associations of address-data pairs that were generated in Step 2. We measure the memory occupancy at the end of this step (which will not change in the next step because the memory is not written to during the reading phase) by noting the number of non-zero elements in the data store weight matrix. It should be noted that using real valued or binary weights makes no difference on the number of zero or non zero elements (1 in the case of binary weights) in the memory, because in each case the same memory locations are written to.

4. **Reading or testing step**: Once all the n associations between the address-data pairs are written, we read in the n address cues one by one to the memory by passing each address vector through the address decoder and the output of the address decoder through the data store. This time the weights of the data store remain unchanged, and the store output vector is calculated by multiplying the address decoder vector with the store weight matrix and taking the d highest elements of the resulting output. This store output vector, consisting of these d maximum indices in order, is compared with the corresponding data vector which was associated to this address vector in step 3. The similarity of these two vectors (the data vector that was associated in step 3 and the present output vector) is found through the usual method of calculating the normalised dot product. We repeat this process for each of the n addresses. For measuring the perfect match capacity, we count the data vector as having been retrieved correctly if the similarity exceeds a predetermined threshold, and total the number of correctly retrieved data vectors out of the n vectors that were written. For

measuring the average quality or average normalised dot product, the similarity values of output and original data vectors for each of the n addresses are added and divided by n to find the average similarity. At the end of this step we have the average similarity over n address-data pairs, as well as the count of how many associations were successfully retrieved.

5. We then **repeat steps 2-4 for many trials**, each time clearing the data store memory completely between trials. We can vary various parameters in different trials and plot the performance of the particular property or aspect of the memory that we are testing on a graph (for example scalability).

4.3.4 Issues in choosing the simulation parameters

Since there are so many parameters in the model, we need to decide which parameters to set to appropriate fixed values and which to vary in each memory experiment. We have tried to make intelligent guesses on the parameters to optimise the memory behaviour, using some insights from the unordered N-of-M SDM studied earlier [27].

For example, we need to **set the values of the significance ratio** used for generating significance vectors and the ratio used for determining the similarity of two vectors. The second significance ratio (for measuring similarity) needs to be kept constant in the experiments, so that we have a uniform standard for measurement of similarity. The threshold we use to measure perfect match (as per the perfect match capacity parameter introduced earlier) also depends on the choice of the significance ratio α, since a low threshold compared to α would allow for more errors to be tolerated in the retrieved vector to certify an output codeword as correct, and vice versa. A high value of significance ratio α is expected to have better memory capacity, since the encoded codewords can be 'packed' more closely in the memory.

An ordered N-of-M code can have two kinds of errors, **errors in choice** (choice of the N elements out of M) and **errors in order**. We would like to penalise errors in choice more than errors in order, if they occur at the same position of significance. This is because having the wrong choice of N neurons firing (out of M) is considered a more serious error (since it would be counted as an error in both the ordered and unordered cases) than having the correct neurons firing but only in the wrong order. Also, since in rank-order codes the earlier spikes to fire

have more effect than later spikes, we would like to penalise choice or order errors at the start more than those towards the end, because the significance of the first neurons to fire is higher in rank-order codes than the significance of later neurons. The second of these conditions has been handled by using the significance vectors with geometrically decreasing significance values, therefore the threshold has to be set keeping in mind the constraint that choice errors should be penalised more than order errors (i.e. the value of the similarity between two ordered N-of-M vectors if some of the components in the N-of-M code of one vector are wrongly chosen should be less than the similarity value if the components are all chosen correctly but the order of some elements is incorrect).

In the N-of-M code for the SDM as a whole, having a very large d (number of activated input neurons in each input word) compared to D is not good for the memory performance, because more weights would have to be set for every codeword written to the data store and so the memory would fill up quickly. The same effect holds for having a large w compared to W.

An unordered memory (where all symbols are encoded as binary N-of-M codes) comes at a lower cost of hardware (because only binary weights are needed), but also has a reduced information efficiency, while an ordered memory (where symbols are encoded using significance vectors) comes at a higher cost of implementation (with real-valued weights) but should give a better information efficiency in bits per synapse. Also, a memory implemented in ordered codes can be tuned in a finer way than unordered codes (because the address space in ordered codes is higher) by changing the matching threshold to recognise if an association has been recovered or read out correctly.

4.3.5 Default parameters in the experiments

In our simulations, the input address and data words are encoded as ordered 11-of-256 codes, the ordering being imposed by taking a significance ratio of α=0.99 and encoding the input as a vector as described earlier. The address decoder layer has 4096 neurons, and the output of the address decoder is encoded as an ordered 16-of-4096 vector. In testing for similarity, we choose a value of 0.9 as the threshold for the normalised dot product in order to count an association as correctly recovered. The reason for selecting 0.9 as the threshold is that it is the value at which we would have at most one choice error (the normalised dot product of two ordered 11-of-256 codes with 0.99 as the significance ratio that

differ in only the choice of the least significant component is 0.91).

In the following sections, we will plot the results of experiments to measure different properties such as scalability and error tolerance of the rank-ordered and unordered N-of-M SDM, with varying memory parameters (such as dimensions of the memory n, N, w and W, significance factor α) and using different parameters to measure the performance as mentioned. In the graphs, we will plot the number of associations correctly recovered (in case of perfect match recovery) or the average dot product (in case of average recovery capacity) on the Y-axis against the numbers of associations written to the memory on the X-axis.

In our graphs we have also plotted the memory occupancy to show how the performance varies as the memory gets progressively filled up. For simplicity we defined the memory occupancy as the number of memory locations that were non-zero (i.e. the number of elements in the data store matrix of the SDM that have been written to). By this definition the occupancy in the ordered and unordered memory should be exactly the same (because the same number of memory locations get written to in both the ordered and unordered cases, and we are counting only the number of locations that are non-zero, rather than the actual values stored in the memory locations), and we can then make a fairer comparison.

4.4 Testing for scalability

For testing the scalability of the memory, we keep the remaining parameters constant and vary only the input size D (in the d-of-D code, whose default value is 11-of-256) by from 128 to 768, keeping d constant. Accordingly, we fill the memory with words from ordered 11-of-128, 11-of-256, 11-of-512, 11-of-768 codes, and check how the perfect match capacity as well as the average recovery or average dot product scales with the input size.

Figure 4.3(a) gives the perfect match capacity scaled with increasing memory size, and figure 4.3(b) gives the average recovery scaled with increasing memory size. We can see that the memory is scalable, since the peak of the perfect match capacity curve rises with the rise in size of the memory, approximately in the same ratio as the memory size.

We then repeated the experiment, this time varying the address decoder size W. The results are plotted in figure 4.4. Here we find the number of words

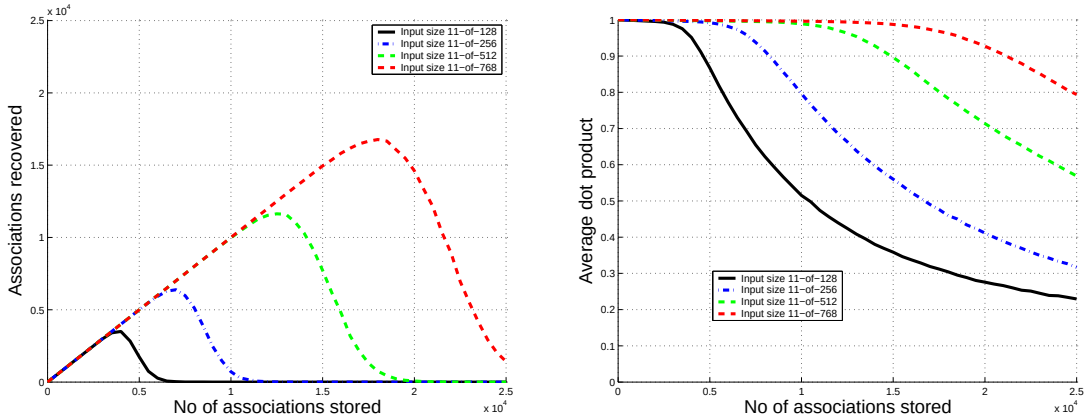

Figure 4.3: (a) Plot of the memory capacity for perfect match (with threshold 0.9) with varying memory size D. (b) Average recovery capacity as memory size increases. The performance is plotted with D as 128(black), 256(blue), 512(green) and 768(red), with d=11 throughout. The significance ratio α for reading or writing to the memory is 0.99 and for measuring output similarity is 0.9. w-of-W is 16-of-4096.

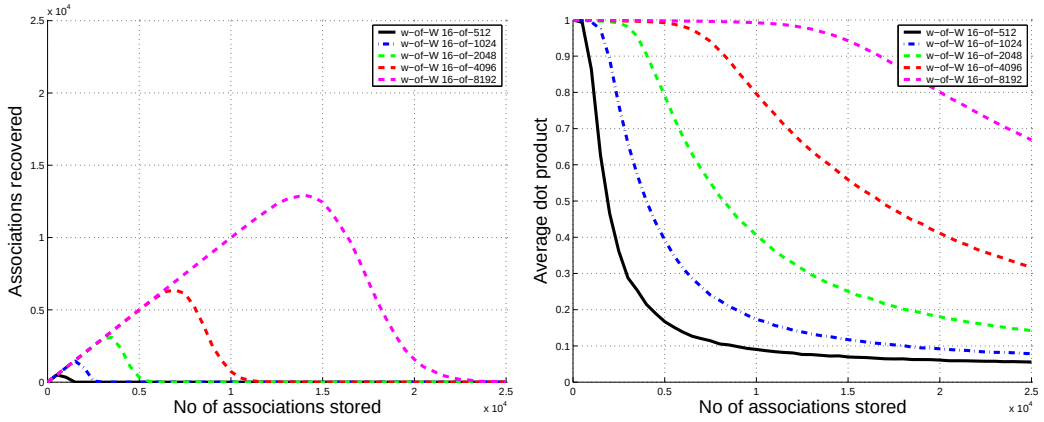

Figure 4.4: (a) Plot of the memory capacity for perfect match (threshold 0.9) with varying address decoder size W. (b) Average recovery capacity as address decoder size increases. The memory performance is plotted with W varying from 512(black), 1024(blue), 2048(green), 4096(red) and 8192(cyan), with w kept as 16. The significance ratio α for reading or writing to the memory is 0.99 and for measuring output similarity is 0.9. d-of-D is 11-of-256.

recovered approximately doubles as the address decoder size doubles. Hence the memory is scalable to the address decoder size.

4.5 Effect of significance ratio α on memory performance

The memory used for this test has same parameters as in the default case. The significance ratio α (also called desensitisation factor) is varied between 1.0 (representing an unordered code), 0.99, 0.9 and 0.5.

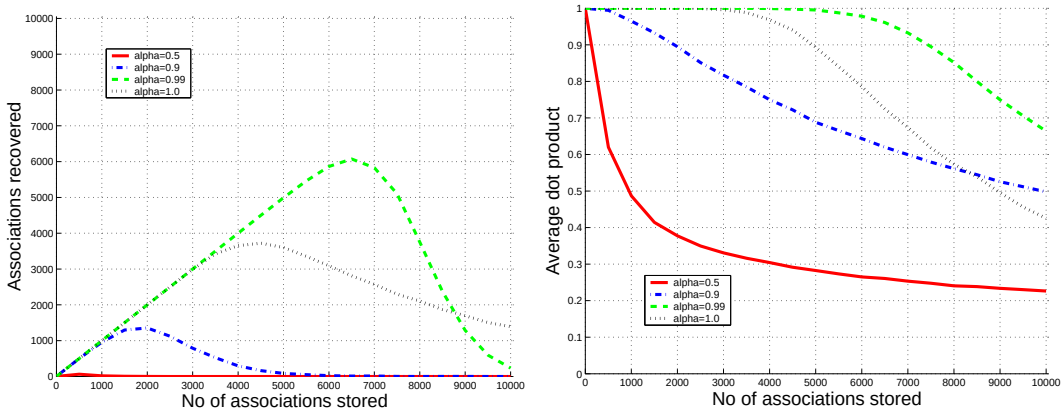

Figure 4.5: (a) Plot of the number of associations recovered (absolute capacity) Vs the associations written to the memory, for different values of the significance ratio α 0.5 (red), 0.9 (blue), 0.99 (green) and 1.0 (black). The significance ratio for measuring similarity is kept constant at 0.99. (b) The recovery capacity of the memory varying with α. The pink line shows the occupancy of the memory.

Figure 4.5 is a plot of how the memory capacity (perfect match capacity) varies with the number of words written to the memory, for different values of the significance ratio α. A value of α=0.99 is used in all cases for measuring the dot product accuracy of the predictions. As we can see, the memory with α=0.5 (in red) performs poorly. The α of 0.5 is low compared to the significance ratio of 0.9 used to calculate the similarity and also the threshold of 0.9 to determine the correct match, which could be the reason for its poor performance. The memory with α= 0.99 (in green) gives the best performance, better than the memory using unordered codes (α=1.0).

4.6 Testing for error tolerance

To test the memory's tolerance to input errors, we use a rank-ordered sparse distributed memory with real weights and standard parameters as described earlier. We first feed it associations (address-data pairs) without any error. Once a certain number of words have been written to the memory using the method described earlier, we insert errors in the input address and see how many data words we get out correctly. Errors in the input address can be either **choice errors** (wrong bits selected out of the d chosen bits following the d-of-D code) or **order errors** (right bits chosen but the order is wrong) or a combination of both. In our experiment we take the least significant bit out of d chosen bits (following an ordered d-of-D code), and set it to a random value between 1 and D, calling it a **bit error**. We introduce one or more bit errors in the addresses that were previously associated to data vectors in the memory, then read out the corresponding outputs from the memory and compare them to the data vectors written earlier. We use the normalised dot product to measure accuracy of the output, and plot both the average accuracy (average normalised dot product) of retrieval and the count of how many words are accurately retrieved over a threshold.

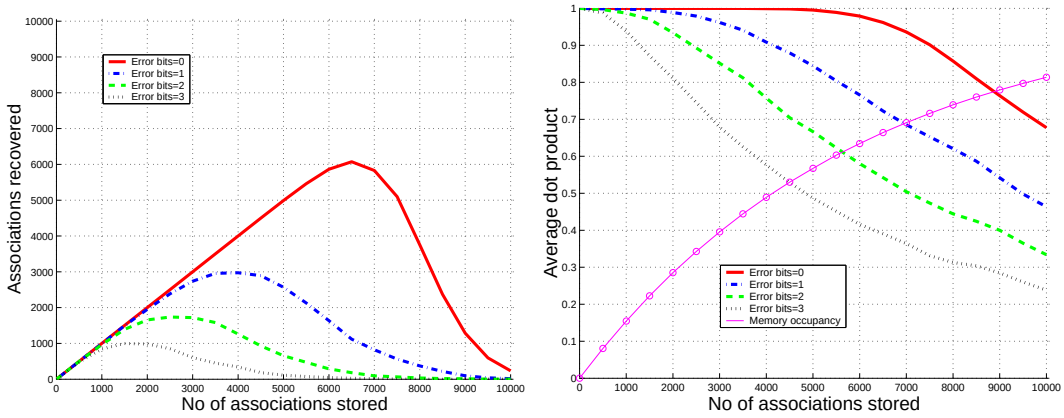

Figure 4.6: (a) Memory capacity (upto 0.9 similarity) and (b) Recovery capacity for varying input error, no error to 3 erroneous input bits. The memory used has real-valued weights and stores ordered ordered N-of-M codes, with α=0.99. d-of-D is 11-of-256, w-of-W is 16-of-4096.

We perform the experiment with the number of bit errors varying from 0 to 3, and plot the absolute capacity and average recovery capacity of the memory. The results of which are shown in figure 4.6.

From the figure, we find that the memory performance degrades when bit-errors are injected, therefore we cannot claim the memory to be error tolerant. However, in the original paper of N-of-M SDMs [27], the unordered version of the memory had been shown to be error tolerant. Perhaps testing with different parameters might give us better results.

4.7 Conclusion

As we can see from the graphs of memory performance, the ordered N-of-M SDM has the following properties:

Scalability: The number of associations correctly recovered scales well with both the SDM memory dimensions D and W. Therefore, the memory can be claimed to be scalable.

Graceful performance degradation: The performance of the memory degrades gradually rather than abruptly when the memory reaches its capacity limit (beyond which writing more words leads to fewer words read out correctly).

Optimal sensitivity factor: Using a sensitivity factor around 0.99 gives the best results in terms of memory capacity.

An ordered memory is more expensive to implement in hardware than an unordered memory because it needs to store real values for the memory connection weights, while the unordered memory needs only a single bit per synapse for the binary weights. This is a weakness of the ordered N-of-M SDM as compared to the unordered N-of-M SDM. However, recent work by Furber et al [26] has shown that it is possible to get an N-of-M SDM with binary weights to store an ordered code, as long as extra memory bits (on top of the memory size required to store the unordered code) are available to store the added rank-order information, and the information is stored with an efficient encoding called rank-order coding, so as to ensure least expense of storage space.

In the next chapter, we shall discuss a framework to update different components of the sequence memory asynchronously, and use that framework (along with the memory described in this chapter) to build the sequence machine.

Chapter 5

Designing a sequence machine

In chapter 2 we introduced the problem of sequence learning and described various approaches to the problem. In this chapter we build a sequence machine that is capable of on-line operation and whose output is a prediction of the following input. We use the sparse distributed memory with rank-ordered N-of-M codes, discussed in chapter 4, to construct the associative memory part of the sequence machine, and combine it with a context memory to build the final sequence machine. In the following two chapters, we shall look at an implementation of this system using spiking neurons.

5.1 A novel asynchronous updating framework for on-line predictive sequence learning

Before we develop our neural model, it is useful for us to define a framework for on-line predictive sequence learning, consisting of the order of presentation of symbols and associations, using which we will develop our algorithm. The purpose is to abstract the finer implementation details and dissect the high-level problem into sequential steps, so regardless of the algorithm or model used for the problem, this framework should be valid.

The associative memory used in the machine should be able to learn the associations between the past history or context and the new encoded input. On being presented with a new input, if the machine recognises the input to be part of a sequence it has seen before (i.e. the association between the present context and the past input has previously been written to the memory), it should

correctly predict the next input (in this case it will not write any new association to the memory, or the writing will not affect the memory significantly), else it will write the association to the memory such that the prediction will be correct the next time the same sequence is presented. The context, which is a function of the past context and the new input, also has to be updated with the presentation of each input, and the output prediction will depend on the updated context rather than the old context.

We can divide the process into the following three steps in temporal order within each timestep, where time t is measured in discrete steps $t\epsilon\{1, 2, .., n, n+1, ..\}$

1. The machine associates the new input symbol I_n at timestep n with the context at the previous timestep C_{n-1}, and writes the association to the associative memory M_n.

$$M_n = W(M_{n-1}, I_n, C_{n-1}) \tag{5.1}$$

   where $W(M, I, C)$ is an update function that takes the value of the memory matrix M_{n-1} at the past timestep (n-1) and outputs the memory at timestep n, which is the result of writing the new association of the new input I_n and the past context C_{n-1}.

2. Based on the input I_n and the context C_{n-1}, the machine creates a new context C_n.

$$C_n = cxt(I_n, C_{n-1}) \tag{5.2}$$

   where $cxt(I, C)$ is an update function that takes in the value of the past context C_{n-1} and the input I_n and outputs the new context C_n.

3. The machine calculates the output by presenting the modified context to the memory.

$$O_n = R(M_n, C_n) \tag{5.3}$$

   where $R(M, C)$ is a read function that reads the memory M_n updated in step 1 and the context C_n updated in step 2 and gives the output O_n at the end of the n^{th} timestep, which is a prediction of the input I_{n+1} during the next timestep.

The above three steps incorporate both prediction (step 3) and learning (step 1), and are a convenient way of dividing the sequence of operations during each discrete timestep n such that the functionality we desire (on-line predictive sequence learning) is implemented.

This framework can also be extended to off-line learning (separate reading and writing phases), simply by fixing the content of the memory at the end of steps 1 and 2 (during the writing phrase), and having only step 3 for the reading phase. By having a special 'end of sequence' input character that clears the old context, we can extend the model to recognise words or groups of symbols in the sequence. However, in this dissertation we have dealt with online learning only.

5.1.1 An example using the framework

Let us assume we are storing the sequence 'ABCABC' in the machine. We expect the machine to learn the sequence 'ABC' the first time and predict the next input letters correctly the second time we present the same sequence 'ABC'. We use this example to illustrate how the memory learns and predicts sequences on-line (i.e. at a single presentation of the sequence), according to the framework described above.

The processing steps are as follows:

1. When we give the first symbol A as input, the initial context output is ϕ (which can either be a special fixed random value or zero) so the association written to the memory is $\phi \rightarrow A$. The new context is a function of the old context and input, which can be represented by cxt(ϕ,A). Let us call this A_1, representing the fact that the value is most influenced by the input symbol A. The predicted output when this new context A_1 is presented to the associative memory (now in read mode) is a function of A_1, which can be written as mem(A_1). Since the memory has never seen such a context before, its prediction will be a random value with no information (let us call this X), or zero (depending on our choice of initial context and the way we implement the context layer).

2. When we input the second symbol B, the association written to the memory is $A_1 \rightarrow B$, which is equivalent to saying that A gives B. The new context becomes a function of old context A_1 and input B, represented by cxt(A_1, B). Let us represent this new context as A_2B_1, denoting that it

is influenced more by the present input B and less by the past input A. Finally, the predicted output on presenting this new context to the associative memory in read mode will be mem(A_2B_1). Since the memory has never seen a similar context before, it will predict another random value, say Y.

3. When we input the C, the association written is $A_2B_1 \rightarrow C$, which is equivalent to saying that A followed by B gives C. The new context is cxt(A_2B_1, C). Let us call this $A_3B_2C_1$ (which is most influenced by C, less by B and least by A). The predicted value by the associative memory in read mode will be mem($A_3B_2C_1$). Here too the memory has not seen this context before, and so will predict another random value, say Z.

4. Now when we input the symbol A to the machine the second time, the memory writes the association between the context and the input. The association written is $A_3B_2C_1 \rightarrow A$. As mentioned earlier, $A_3B_2C_1$ denotes that the context is influenced most by C, less by B and least by A, so this indicates that A followed by B followed by C gives an A, or approximately, C gives A. The new context formed from the old context and the input will be $cxt(A_3B_2C_1, A)$ which will be approximately $B_3C_2A_1$. On presenting this to the memory, the output of the memory will be a function of the present context, represented as $mem(B_3C_2A_1)$. Now since it had learnt previously in step 2 the association $A_1 \rightarrow B$, and this present context $B_3C_2A_1$ is most influenced by A, it will correctly predict the next input to be B.

5. When we give B the second time, the same procedure is repeated. The association written is $B_3C_2A_1 \rightarrow B$. The new context is $cxt(B_3C_2A_1, B)$ which can be represented as $C_3A_2B_1$ representing the relative influence of A, B and C. The output of the associative memory on presentation of this new context will be $mem(C_3A_2B_1)$ which will be C (because the machine has learnt previously in step 3 that A followed by B gives C).

6. Finally, when we give C the second time, the association it writes is $C_3A_2B \rightarrow C$, the new context is cxt(C_3A_2B, C) which will be A_3B_2C. The predicted next input is given by mem(A_3B_2C), which will be A, based on the association written to the memory in step 4.

Thus we can see how the sequence machine would predict the future B,C.A

(in steps 4,5,6) inputs correctly after a single presentation of the sequence A,B,C (in steps 1,2,3).

5.2 Encoding the context: Combined model

In chapter 2, we discussed the context neural layer and shift register models as ways to create the context as a function of the past context and the input. Both these models both have their advantages and disadvantages, which we discussed in chapter 2. Our goal is to have a model that combines the best of the two and minimises their defects.

In creating a new context, our aim is that it should not be overly influenced by past inputs, but should still be able to remember enough of the past to discriminate the context when 'locking on' to a sequence it has recognised (in order to predict the new symbol in the sequence). The chosen model should be at least as good as a shift register, and perhaps a bit better, in the sense that forgetting of the past context or look-back should be gradual, rather than a shift register where forgetting of the past (beyond its time window) is abrupt.

We create a model that combines the advantages of both the earlier models by having a modulation factor which can be tuned as a knob to produce the effect of both systems, and facilitate a smooth transition between them. In doing so, we expect that for some value of the factor which is between these two systems, we can combine the best features of both.

The modulation factor λ is a mechanism by which we can control the influence of the past history in determining the prediction of the next symbol. The effect of the context is modulated by this factor. For a certain range of λ, we can ensure that the past history is slowly forgotten, and the present inputs have a greater role than the past history in predicting the next inputs. For another range of λ, the effect of the past history will be predominant in the prediction. For an intermediate range, the prediction will depend somewhat on the past and somewhat on the present inputs.

We combine the shift register and neural layer by using a separate context layer with this modulated context.

Figure 5.1 shows how the new context is formed from the old context and input.

The context is modified in the following two steps:

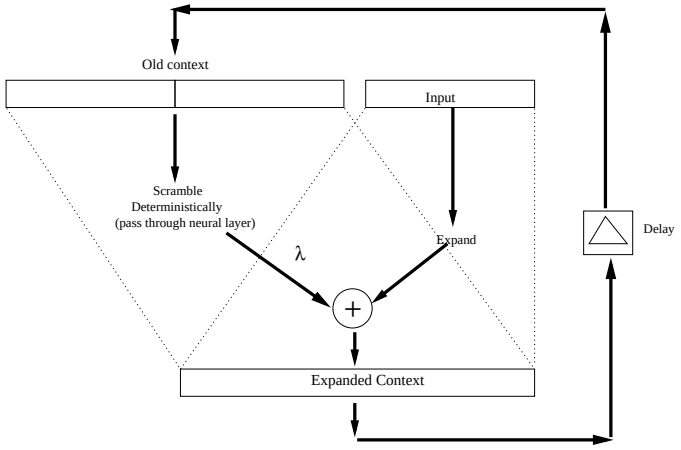

Figure 5.1: Creation of the new context from the old context and the input, in the combined model.

1. The old context is **projected into high dimensional space**, by passing it through a neural layer with fixed weights.

2. The projected version of the old context is then **multiplied by the context modulation factor** (λ) which is less than 1, and is then **added to the expanded input code** (which has been expanded by another projective function to produce a rank ordered N-of-M vector of the same size as the context), and **K maximum components of the sum** are chosen to make the new context. This ensures that the most important bits of the input replace those of the old context, and the old context bits get shifted to less important bits of the new context. Thus the input bits are shifted up and context bits down. This part represents a shift register.

If the value of λ is 0, the new context will be the same as the new input, and this will be like a pure associative memory where the past history plays no role in determining the next predicted symbol. As the factor is slowly increased, the effect of the context increases. Since we use rank-ordered N-of-M codes with the vector of significances as described earlier, keeping the scaling factor equal to α^{N+1}, where α is the significance ratio, will ensure that the most important bit of the old context is given a weight less than the least important bit of the new context, thus enabling it to forget the old context gradually, and the expansion acts as a shift register in that the old context is 'shifted' to the less important bit of the new context and the present input takes prominence in determining the value of the new context. As the scaling factor is further increased, the relative

importance of the old context increases, until the old context becomes equally important to the present input (similar to a pure neural layer) at λ=1.

5.2.1 A mathematical description of the combined model

As per the notation used earlier, I_n represents the input vector of dimension [D,1] at the end of discrete timestep n and C_n represents the context vector of dimension [M,1] at the end of the n^{th} timestep. Since we are using ordered N-of-M codes in our memory model, I_n is encoded as an ordered d-of-D code and C_n as an ordered m-of-M code.

In the combined model, the context C_n at timestep n can be represented as:

$$C_n = nofm(\lambda scale(P_1 C_{n-1}) + scale(P_2 I_n), m, M, \alpha) \tag{5.4}$$

Where

scale is the scaling function, scaling the vectors such that the sum of components equals 1.

P_1 is the weight matrix of dimension [M,M] connecting the context output of the previous timestep to the context layer.

P_2 is the weight matrix of dimension [M,D] connecting the input to the context layer.

$nofm$ is the function representing the transformation of the result as an ordered m-of-M vector with significance ratio α.

In this model, a value of $\lambda = 1.0$ implies that the present input as equal in importance to the past context in determining the new context value, and $\lambda = 0.0$ implies that the new context is determined purely by the present input, without any importance given to the past context. A value of $\lambda > 1.0$ means that the present input is more important than the past context in determining the new context.

5.2.2 Context sensitivity factor as a convex combination

An alternative way to encode the context could be to use a **convex combination** of the input and context. Let the context sensitivity parameter in this case be represented by the symbol Λ. Hence the expression for the context C_n at the end of timestep n is as follows :

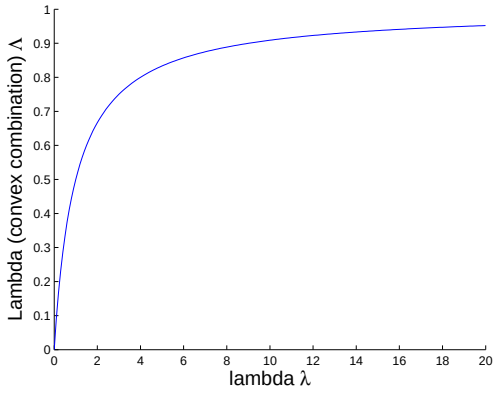

Figure 5.2: Relationship between λ and Λ

$$C_n = nofm(\Lambda scale(P_1 C_{n-1}) + (1-\Lambda) scale(P_2 I_n), m, M, \alpha) \tag{5.5}$$

Here, $0 \leq \Lambda \leq 1$

Such an arrangement, for the context sensitivity parameter Λ=1.0, means that the new context is purely a function of the past context (no influence of the input). $\Lambda = 0$ represents the case where the new context is determined purely by the present input (no influence of the past context). $Lambda = 0.5$ represents the case when the present input and context are equally important in determining the new context (equivalent to λ=1.0 in the previous model)

The relative influence of the context and input is $\lambda : 1$ in the first combined model and $\Lambda : (1-\Lambda)$ in the convex combination model. The relationship between λ and Λ is as follows :

$$\lambda = \frac{\Lambda}{1-\Lambda} \tag{5.6}$$

$$\Lambda = \frac{\lambda}{1+\lambda} (when \Lambda \neq 1.0) \tag{5.7}$$

$\Lambda = 1.0$ represents the case where the influence of the input is negligible in determining the new context, and is equivalent to an infinitely large λ. In effect, Λ is equivalent to a squashing function on λ, limiting its allowable range from $[0, \infty)$ to the range $[0, 1]$. The relationship between λ and Λ is illustrated in figure 5.2.

In our tests of the sequence machine performance described in chapter 8, we shall investigate both of these models.

5.3 The complete sequence machine

By using the rank-ordered N-of-M SDM as described in chapter 4 as the associative memory, and using the encoding of the context as described earlier in this chapter (combining the neural layer and shift register models of context encoding) we have the complete model of the sequence machine. We also need to have an encoder to encode the input symbols (which would be 1-of-A code, representing one symbol from the alphabet of A symbols) to the appropriate ordered N-of-M code and a decoder to decode the output of the associative memory, which is an ordered N-of-M code, to a 1-of-A code representing the output symbol which is a prediction for the next input. The sequence machine thus developed will have both short term memory (represented by the dynamic representation of the context in the context layer) and long term memory (represented by the data store layer of the SDM, which is a memory of finite size into which the associations are written).

Thus, the complete sequence machine has three primary components: an encoder, the neural sequence memory, and a decoder. The characters of the sequence are fed, one at a time, into the encoder. The encoder has a unique encoding for each character in the input alphabet. If implemented in neurons, the encoder can be represented by a single neural layer having fixed weights, such that the mappings of the characters to the neural code are fixed: it behaves like a lookup table. The decoder is similar to the encoder, except that the inputs and outputs are reversed.

As described in chapter 4, the associative memory used in the sequence machine has real-valued weights and associates the context cue with the input symbol. It is made of two layers, the address decoder and the data memory, as in the N-of-M SDM. The context layer is separate from the associative memory. It is also made of neurons and has fixed weights, and implements the combination of neural layer and shift register. We use the max function as the training algorithm for the weights in the data store layer of the ordered N-of-M SDM. Learning (change of weights) takes place in the data store layer of the SDM only, and the weights of the context layer and the address decoder layer remain fixed.

The figure 5.3 gives the structure of the complete sequence machine showing its various components.

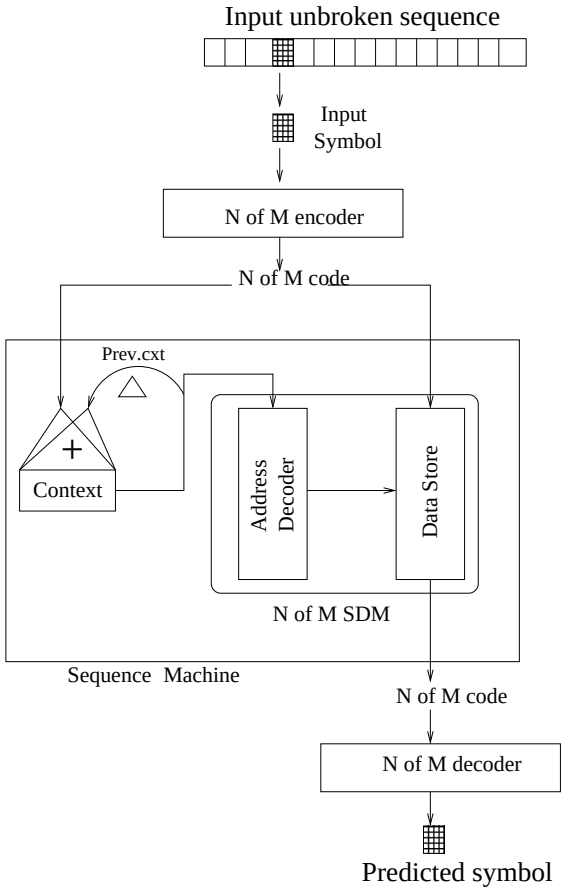

Figure 5.3: Structure of the complete sequence machine having the SDM memory (composed of address decoder and data store), context, encoder and decoder layers

5.3.1 Operation of the sequence machine

The operation of the sequence machine during each discrete timestep n can be described by the following algorithm:

1. Encode the input symbol I_n (a vector of 1-of-A code representing a chosen symbol from the alphabet A) as an ordered d-of-D code I'_n by passing it through an N-of-M encoder.

2. Feed the resulting ordered d-of-D code I'_n to the data store layer of the SDM (at the data inputs) as well as to the context layer.

3. Feed the delayed context layer output C_{n-1} from the previous timestep into the address decoder layer of the SDM, and to the context layer (via feedback connections). The address decoder takes the old context C_{n-1} as input and calculates the ordered w-of-W decoded address W_{n-1}.

4. Input the address decoder output W_{n-1} to the data store.

5. **Write the association** of the address decoder output W_{n-1} and the encoded input I'_n to the data store layer (by setting its weight matrix to the max of the outer products of the vectors and the previous matrix value) .

6. **Generate the new context** C_n in the context layer from the previous context C_{n-1} and the ordered d-of-D input code I'_n.

7. In **reading from the SDM**, feed the new context C_n into the address decoder, which generates a w-of-W coded output W_n. Feed the output W_n into the data store.

8. The data store, in the read mode, reads the address W_n and computes the output O'_n as an ordered d-of-D code.

9. The ordered d-of-D code O'_n is passed through an N-of-M decoder (whose weights are transpose of the encoder used in step 1), which generates the output O_n as a 1-of-A code, representing the chosen output symbol from alphabet A, which is a prediction of I_{n+1}.

Thus, the whole operation proceeds discretely with each input character.

5.4 Conclusion

In this chapter, we described a framework to describe on-line predictive sequence learning, and designed an on-line sequence machine based on that framework, using the combined model of encoding the context. The sequence machine described operates with time-abstracted significance vectors to encode outputs of various components.

In a model with spiking neurons, we have to work with spike timings, and the processing becomes more complicated than simply calculating products of vectors and matrices. In the next chapter, we shall consider issues in implementation with spiking neurons.

Chapter 6

Modelling with spiking neurons

In the previous chapter the implementation of the sequence machine was described assuming the abstraction of time through the use of significance vectors. In this chapter and the next, the implementation of the machine using spiking neurons will be explored.

The principal aim of this dissertation is to use low-level spiking neural components to build a sequence machine. In doing so, we need to consider a number of problems, such as how to keep spike bursts, which are emitted by different layers and encode symbols, well separated in time. This is needed if we are to have stable dynamics to make sense of them as coherent symbols and prevent inter-symbol interference. We look at the use of feed-forward and feedback inhibition to solve some of these problems. Our intention is to show that it is feasible to have a system that can transmit a coherent and stable burst of spikes across different neural layers.

In this chapter we consider issues that are commonly encountered in modelling spiking neural systems in general (with proposed solutions), and in the next chapter we shall examine issues related to the sequence machine, especially the integration of various neural layer components to form a coherent working system.

6.1 Issues in spiking implementation

In implementing the sequence machine using low-level asynchronous spiking neurons, there are some issues we need to consider. These issues can be grouped into the following types:

1. General issues connected with modelling any high-level function using asynchronous spiking neural systems (including implementing the encoding scheme in 6.2.1, defining the beginning and end of a burst in subsection 6.2.2, and issues relating to the burst coherence and stability in subsection 6.2.3)

2. Issues concerning the implementation of different components of the system using spiking neurons (including implementing the learning rule in subsection 6.2.4)

3. Issues regarding the functioning of the sequence machine as a whole (including timing issues such as keeping timing dependencies between components in subsection 6.2.5)

4. Issues connected with the spiking neural model (including choice of a suitable model to implement the desired functionality in subsection 6.2.6).

In the following subsections, we discuss the different issues and suggest our solutions to enable the system to work correctly.

6.1.1 Implementing the rank-ordered ordered N-of-M encoding scheme

First of all, we need to specify how to treat the spike bursts transmitting information in the system implemented with spiking neurons, and understand their relation with the time-abstracted symbolic system (described in chapter 4) we have been using so far in the SDM memory and in the sequence machine, in which we considered the symbols to be encoded as ordered N-of-M significance vectors. With the system implemented in spiking neurons, we assume that the symbols are encoded as bursts of spikes emitted by neural layers. The output burst from one neural layer is input to the next layer, which generates another burst on its output, and so on. The ordered N-of-M code is followed in the spike bursts, meaning that code transmitted is in the choice and relative temporal order of firing of N spikes out of M neurons in the layer forming a burst. We assume that a spike emitted by a neuron is immediately transported to the inputs of all neurons that it is connected to. However, we do have delays between the input and output spikes of a neuron, that are caused due to the dynamics of the chosen neural model (described in a later section), and use neurons in our systems that utilise such delays to synchronise different bursts.

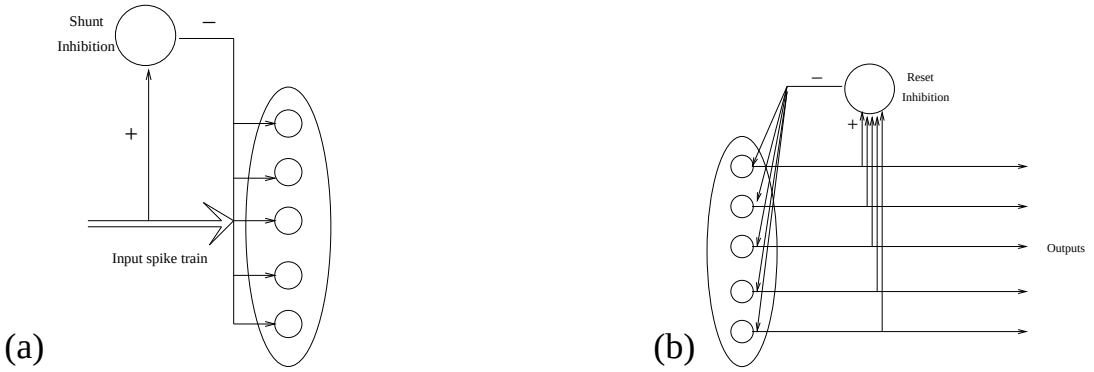

Figure 6.1: (a) Feed-forward shunt inhibition to make a layer sensitive to a temporal order of input spikes. (b) Imposing an N-of-M code in a layer of spiking neurons through feedback reset inhibition to control spiking activity when a certain number of output spikes have fired.

The coding scheme we are implementing is rank-ordered N-of-M code, which is transmitted as the choice as well as the rank of spikes in a burst. Such a code can be generated by having a feedback reset inhibition neuron that takes inputs from the M outputs of the layer and feeds back to the neurons in the layer. The feedback neuron acts as a counter and has a threshold of N. When N output spikes have fired in that layer, this feedback neuron fires a resetting spike that resets all the neurons in the layer, thus ensuring that exactly N neurons can fire spikes in a burst. We assume here that two neurons do not fire at exactly the same time, or the feedback neuron has a deterministic way to choose an arbitrary order between two simultaneous outputs. We only need to make sure that the output order of spikes from a layer in response to a given input order is equivalent in the case of the time-abstracted and spiking neural models.

A neuron or a layer can be made sensitive to a particular order of spikes by employing a feed-forward shunt inhibition neuron (FFSI) or a population of interneurons, following the method used in Thorpe's model [77, 22, 76] to desensitise a layer with each additional spike in the burst. The shunt inhibition neuron, which has a different time constant than the neuron it inhibits, takes inputs from all the inputs to the layer (to which it is connected as a shunt) and outputs an inhibitory spike to all the neurons in that layer.

A quantity called input sensitivity or significance of a neural layer is defined as the effect on the activation increase of the neurons of a layer in response to input spikes to that layer. The spike emitted by the FFSI neuron decreases this quantity. This inhibitory spike can decrease the significance in two ways: additively or

multiplicatively. If we assume it to act additively, the inhibitory spike to a neuron would set its sensitivity down by a fixed value with each additional input spike. If the effect of the FFSI spike is multiplicative, each inhibitory spike causes the sensitivity of the neuron to be multiplicatively reduced by a fixed constant. The multiplicative sensitivity can be considered a special case of the additive if we consider decaying activations and equally spaced input spikes [26], so we shall be using it in our spiking implementation.

Therefore, each inhibitory spike decreases the sensitivity of a layer multiplicatively by a constant ratio α, which is less than 1 and which we call the desensitisation factor. Decreasing the sensitivity by a constant ratio α each time makes the rise of the activation of the neuron in response to future input spikes reduce by the same ratio. The FFSI neuron is different from common spiking neuron models such as LIF [53] in the sense that the effect of its spikes is multiplicative rather than additive, and it affects the significance value of the neurons in the layer rather than the activation.

The FFSI neuron fires an inhibitory spike to all the neurons in the layer with each successive input spike. The inhibitory spike sets the input sensitivity of each neuron in the layer to the same amount, regardless of whether the neuron receives the input spike. Since the first input spike strikes the layer directly before the shunt inhibitory spike has time to decrease the sensitivity, the first spike will have maximum effect (due to absence of inhibition) in increasing the activation. The shunt inhibition spike strikes the layer immediately after and multiplicatively decreases the sensitivity of all neurons in the layer to α, therefore the second input spike will have its effect decreased by a ratio of α, the third input spike will have an effect proportional to α^2 and so on. This is similar to the vector of significances [1, α, α^2, ...] that we used in chapter 4 to encode symbols in the time-abstracted model.

Thus, a rank-ordered neural layer can be implemented by a combination of feed-forward shunt inhibition and feedback reset inhibition, as illustrated in figure 6.1.

6.1.2 Beginning and end of a burst

One of the questions we need to consider when implementing such a system is the identity of the burst of spikes. Since the choice and order of firing of the neurons in a burst of spikes represents the code being transmitted, we have to

be careful not to let errors such as extra spikes from a different burst or wrong timing order creep in. Therefore, we need to define how to identify the beginning or end of a burst. Since we have stipulated that all bursts follow the N-of-M code, we can have counter neurons to count the number of spikes at the output of a layer. We make the convention that the first spike of an input burst signals the start of the burst, thus activating all neurons in the layer to receive the input code being transmitted. The end of an output burst is signalled by the output counter neuron connected to all the outputs of the layer firing a resetting spike when a total of N output spikes have fired, which resets all the neurons in the layer.

6.1.3 Coherence and stability of the burst

To have a well-behaved system, we need to ensure that the spike bursts which transmit the code are stable and coherent. This means the following things:

1. **The intra-burst separation (the temporal separation between the first and last spikes within a burst) should be within bounds** as the burst propagates through the system. It should not progressively increase or decrease as the bursts travel through successive neural layers in the system, else the bursts may interfere with other neighbouring bursts in the system (leading to inter-symbol interference). Specifically, the intra-burst separation should be small compared to the inter-burst separation, to enable the system to distinguish different bursts clearly. The bursts should be coherent and their integrity should be maintained as they pass though the system.

2. **The bursts should not fade out.** Since the bursts are regenerated by each layer, there is a danger that there may be fewer spikes in the output burst of a layer than in the input burst, and after a few layers the spikes may die out completely, since we need a certain degree of neural activity to sustain the burst.

3. **The spike activity should not increase with each successive layer till it reaches a very high level.** This is the reverse of the fading out phenomenon. The number of spikes constituting the burst can increase as the burst propagates through successive layers, leading to an unacceptably high level of spiking activity and destroying the code being propagated.

4. **The code being propagated should be correct.** This means that the order of firings should be computed correctly in each burst that is propagated. An input spike burst to a neural layer should produce an output burst corresponding the output significance vector in the time-abstracted model.

The stability problem can also be considered in a wider context of control theory [70], where stability is a standard problem. To have stability in a system, some mechanism is needed to control the output (usually implemented through negative feedback). In our case where we have a system with multiple neural layers, since we have the problem of controlling the burst activity within a specified level, we need to have some inhibition mechanism to control the output spiking activity of the layer.

In a later section, we will implement a neural system that has the property of stable propagation of burst activity through a feed-forward network.

6.1.4 Timing issues

In the sequence machine design, there are strict timing requirements between the symbols. For example, since the output of the system is a prediction for the next input, it is essential that the outputs of the sequence machine fire before the next inputs arrive.

Another timing problem is the control and synchronisation of the firing times of spike bursts across different layers, in order to fine-tune the system to perform a specific function. For example, the sequence machine has two kinds of inputs to the context layer, one of which is the fed back previous context from the delay layer and the other is the present encoded input from the encoder layer. If the feedback time of the old context through the delay layer is too fast compared to the time separation between different waves of inputs, the context spikes could circulate around the feedback loop before the next inputs come in, thus spoiling the code to be transmitted. Therefore, it is important that both inputs to the context layer arrive at approximately the same time. The delay layer latency and the time gap between successive inputs can be matched to fulfill this requirement.

We also need to make sure that the neurons in a layer do not fire until all the inputs to that layer have arrived. If they start firing before all the inputs have arrived, the code transmitted, which is based on the timing order of the spikes,

will not be correct.

In our implementation with spiking neurons, we have met the mentioned timing constraints by inserting extra delays through delay layers whose neurons have long time constants. The delays inserted by these delay layers are only due to the internal dynamics of the neurons, because of their long time constants. By engineering the time constants to a suitable value, we have tuned the system to perform in a precise way keeping all the required timing constraints.

6.1.5 Implementing the learning rule in spiking neurons

For the learning rule in the correlation matrix memory component of the sequence machine, which is part of the rank-ordered N-of-M SDM implementation, we used the max function as described in chapter 4. In the time-abstracted model, the association of two significance vectors representing the address and the data was written to the memory as follows: the new weight matrix is the maximum of the old weight matrix and the outer product of the significance vectors (refer to the formula in section 4.2 of chapter 4).

Implementing this algorithm in spiking neurons presents a problem: the two bursts of spikes from the learning inputs (the data inputs of the SDM) and normal inputs (the address inputs of the SDM) do not arrive at the same time. In the sequence machine design, the old context spikes (from the previous wave) which arrive earlier are to be associated with the present encoded input spikes, which arrive later. Therefore the significance vector representing the order of spikes in the burst corresponding to the old context must somehow be stored until the learning inputs arrive in the next wave. We accomplish this by storing the significances in the synapses of the neurons of the data store layer. This is only possible if we assume the neurons in the layer to have a synaptic memory, which can store the respective significance vector components to associate with the components of the learning input spikes. We can think of this synaptic storage as equivalent to an eligibility trace (mentioned in chapter 3), which is used to determine the change of connection weight when the other part of the association arrives later. These eligibility traces stored in the synapses of the neurons have to be cleared when the writing of the association to the memory is concluded, in order to prepare for the next burst.

The local neurons obtain the global information of the rank of the inputs (in order to calculate the significance components to store in the synapses) through

the FFSI neurons, that keep track of the current rank of the firing in the form of the sensitivity or significance values. When an input spike to a neuron fires, it sets its sensitivity value in the synaptic memory to the current value of the FFSI significance representing its firing rank. In this way the shunt inhibition acts as a mechanism to communicate global order information to the individual neurons in the neural layer.

Another problem is implementing the max function in the learning rule (the new weight component being the maximum of the old weight and the product of the respective components of the significance vectors being associated) using spiking neurons. We can treat the max function as a dynamic threshold set to the present value of the connection weight, the connection weight being written or modified only if the product of the eligibility trace (i.e. the significance vector component) of the old context and the significance of the encoded new input is above this threshold. Therefore, the STDP-style learning rule corresponding to the max function in the SDM is as follows: The data inputs serve as the teaching signal, and the weight change is a nonlinear function of the old weight, input eligibility and teacher signal, such that the max function is implemented (i.e. a change of connection weights happens only if the product of the corresponding significance components is more than the existing connection weight, and the new weight is set to this product).

6.1.6 Choosing a spiking neuron model

In this subsection we describe our criteria for choosing a suitable model of spiking neuron.

We need the simplest, fastest and most flexible spiking neural model that can achieve the functionality we require, which is to model the dynamical behaviour of the sequence machine. We want a model that can implement the ordered N-of-M code and can produce an output equivalent to the time-abstracted model using significance vectors. We are modelling using point spiking neurons (neurons without spatial properties) that make the decision to fire based only on local information of the input spikes, not on any global variables. The guidelines and assumptions regarding the spiking neuron model implementing the system are summarised below.

- The spiking neural model must be biologically plausible. In other words,

any assumptions we make regarding the model ought to be at least partly justifiable in the behaviour of biological neurons.

- The decision of a neuron to fire must be based on its input spikes only. Therefore, any global parameters in the system, such as the desensitisation factor, will also have to appear to each neuron as a separate local input spike.

- The neuron fires when an internal quantity of the neuron known as activation exceeds a threshold. The threshold can be fixed or dynamic. The activation represents the integration of the effect of all the input spikes to the neuron, weighed by the strength of the input connections. The integration function may be nonlinear.

- All spikes are assumed to be identical. The only information in them is in their time of firing.

- As a simplification, we assume that the neurons are point neurons, i.e. they have no spatial properties (such as are used in compartmental models of spiking neurons [48]).

- The neuron can have different kinds of inputs, i.e. it can have input spikes from different layers, and can distinguish between them if necessary. This is needed to model the context layer of the sequence machine, which is fed both the current input and the fed back past context through a delay layer.

- The neuron may have delays built into it, i.e. a time delay between the input and output spike due to its internal dynamics. However, we assume the neuron to have no wire delays in the transmission of the spikes to the next layer. The output spikes from one layer appear immediately to the inputs of the next layer connected to it.

- The neuron can have some special inputs, which it can be programmed to treat differently from normal inputs. For example, there can be some inputs which are meant for learning, by modifying the connection weights.

We will need the spiking neuron model to be able to model properties such as sensitivity to inputs (in order to implement rank-order codes), the ability to write an association of two rank ordered codes to the memory in the form of

connection weights (to implement the learning rule), etc. The model should be able to implement the functionality of the sequence machine network (including feedback layers such as from the context layer) and to produce an equivalent effect to the time-abstracted model with significance vectors.

In the next sections, we will describe two spiking neuron models that fulfill the above basic requirements, and examine their suitability for modelling our system.

6.2 RDLIF model

The rate-driven leaky integrate and fire (RDLIF) model is a variant of the standard leaky integrate and fire (LIF) model [53]. The only difference is that incoming spikes in the standard leaky integrate and fire model increment the activation value, while in the RDLIF model they increment the rate or the first derivative of the activation value.

In this model, the behaviour of a neuron can be described by two variables: the activation, a, which is the quantity which induces the neuron to emit a spike if it exceeds a threshold, and the activation driving force, r, which controls the rate of increase of the activation. Both activation and driving force decay with time, and the rate of decay is governed by their respective time constants τ_a and τ_r. Here, the activation is a dimensionless variable, the time constants have dimensions of time, and the activation driving force has the dimension of one over time.

The driving force of the i^{th} neuron increases with incoming spikes and decays with time t, until it reaches a resting value r_0.

$$r_i(0) = r_0 \tag{6.1}$$

$$\frac{dr_i}{dt} = \dot{r_i} = \sum_j w_{ij} x_j - (r_i - r_0)/\tau_r \tag{6.2}$$

Here $x_j = \sum_n \delta(t - t_n)$ is the sum total of impulse functions (i.e. integrating this with time will give the total area under the impulse curves representing the spikes) of the input spikes emitted from the jth input neuron and w_{ij} is the connection strength.

The driving force drives the activation, which itself decays with time to a resting value a_0.

$$\frac{da_i}{dt} = \dot{a_i} = r_i - (a_i - a_0)/\tau_a \tag{6.3}$$

If the activation of the neuron exceeds its local threshold, it fires a spike and immediately its activation is reset to a refractory level, and its driving force is reset to r_0 to prevent the activation from increasing.

$$a_i >= \Theta_i => (y_i = \delta(t - now)) \wedge (a_i = a_{ref}, r_i = r_0) \tag{6.4}$$

Solving the above equations, we arrive at the following expression for activation of a neuron having a single input spike at time t=0.

$$a = Ae^{-t/\tau_a}(1 - e^{-t/\kappa}) \tag{6.5}$$

where $\kappa = \tau_a\tau_r/(\tau_a - \tau_r)$ and $A = r_0\kappa$ are constants.

If $\tau_r = \tau_a = \tau$, and asymptotic values r_0 and a_0 are set to 0, and all the weights are normalised to 1 i.e. the increase in the activation rate due to an impulse input spike is 1, for a single RDLIF neuron that has an input spike at time t=0, the expressions for the four state variables of the neuron: $r, \dot{r}, a, \dot{a}$ are:

$$r = e^{-t/\tau} \tag{6.6}$$

$$\frac{dr}{dt} = \dot{r} = -1/\tau e^{-t/\tau} \tag{6.7}$$

$$a = te^{-t/\tau} \tag{6.8}$$

$$\frac{da}{dt} = \dot{a} = e^{-t/\tau}(1 - t/\tau) \tag{6.9}$$

Figure 6.2(a) shows the shape of the activation curve following a single input spike at time 0. We see that the activation at first increases due to the increased driving force caused by the incoming spike, but after a time the decay becomes dominant. For an RDLIF neuron to fire, it should get a sufficient number of input spikes within a specified time to enable it to reach the threshold before it 'dies out' because of the decay of the activation and activation rate. There is an inherent time lag between the input spike and the maximum activation reached by the neuron. Such an intrinsic delay between an input and output spike is

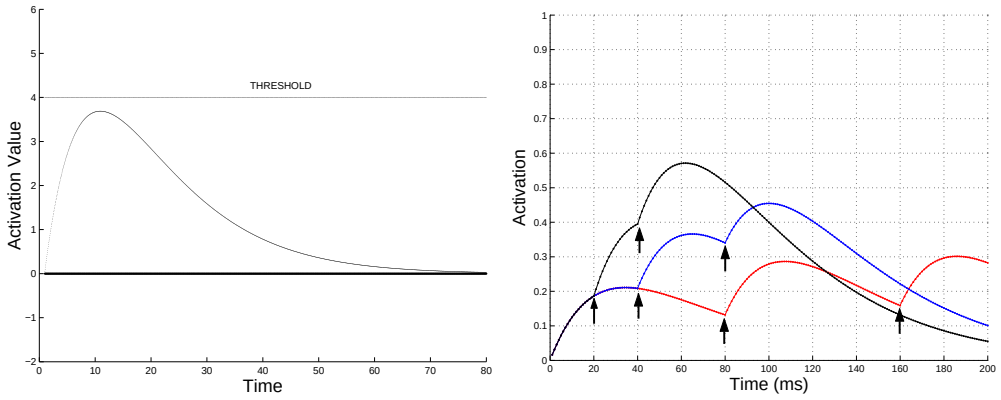

Figure 6.2: (a)The plot of a typical RDLIF neural activation with time, when the neuron receives a single input spike at time t=0.(b)Plot of activations of three identical RDLIF neurons which get input spikes at 20,40 msec (shown in black), 40,80 msec (shown in blue), 80,160 msec (shown in red). The neuron (with black activation) which gets most closely spaced input spikes has the highest activation.

useful in a feedback system to stop potential interference between different spike bursts. Without such a delay (as in a standard leaky integrate and fire neural model [53]) at least one output spike from each neural layer would have to fire at the same time as an input spike from that layer (to enable a neuron in that layer to cross the threshold and fire the last spike of that burst), which can cause problems if a feedback loop exists in the system (since this condition holds true for each neural layer in the loop).

Another characteristic of the RDLIF model is that closely occurring input spikes reach the threshold quicker than those spread out, because if the average inter-spike interval is greater, the decay becomes more important. This is illustrated in figure 6.2(b).

6.2.1 Comparison of the RDLIF model with the standard models

The standard leaky integrate and fire (LIF) model [53] can be considered a special case of our rate driven LIF model, if the activation rate time constant is very low and the activation time constant is very high. In addition, it is possible to model characteristics such as habituation (decreased response to a repeated stimulus), sensitization (heightened response after a painful stimulus), refractoriness (the neuron not being able to fire another spike immediately after spiking), etc in this

model.

It can also be considered as an implementation of Gerstner's spike response model (SRM) [28], although the SRM model is more general. The kernel function a(t) in response to an input in the RDLIF model implements the time course of the response to an input spike ϵ in the SRM model. The family of curves generated by the above equations (when the time constants of rate and activation decay are varied) is very similar to the family of curves of the build-up of the activation in response to an input spike in the SRM model.

It should also be mentioned that the RDLIF model is more biologically plausible than the standard LIF model. The conductance of the cell membrane of the neuron can be thought of as representing the rate of activation, as it passes through a resistor (the resistance of the cell as represented by the ion channels and neurotransmitters) and the current is similar to the activation.

6.2.2 Suitability of the RDLIF model for implementing the time-abstracted model

We need to make sure that the effect of an input burst of spikes on a layer of neurons (i.e. the order of spikes in the output burst of that layer) is equivalent to the effect in the time-abstracted model (which is the order of output spikes as can be deduced from the output significance vector). In the time-abstracted model, each symbol is encoded as a vector of significances in the geometric progression 1, α,α^2 etc, where α is the significance factor.

In the RDLIF model, the increase of activation of the neuron in response to an input spike is a function of the time of the input spike, as well as the connection weight and the input sensitivity. This is because the input spike increases the rate of the activation, which is the slope in the activation-time graph, rather than the activation itself. In the time-abstracted model, the increase of activation in response to an input spike is a function of the connection weight and the input sensitivity alone. Therefore, the addition of time in the equation of the activation makes it difficult to ensure that both the models will behave identically and will give the same output order of spikes in response to a given input order.

An analysis of a simple two-layered network with two RDLIF neurons in each layer, in which the outputs of the time-abstracted and timed models have been

compared, can be found in Appendix A. As we see from the appendix, the condition for the model implemented with RDLIF neurons to perform equivalently to the time-abstracted model is a function of the connection weights and the significance ratio, and tuning the two models to work identically will require us to tune these parameters for each combination of the input and output firing order. Therefore, generally speaking, implementing the temporal abstraction by the RDLIF model of spiking neuron is not feasible. We have to choose an alternative spiking model which can perform exactly as the abstracted model. One such model with simpler dynamics is described in the next section.

6.3 The Wheel model

The wheel or spin model of a spiking neuron is a simple linear model with spontaneous linear activation (say as a result of a tonic input) and no activation leakage, in which the neuron can be visualised as a spinning wheel. This model is also known as the firefly model, and was originally proposed to model the synchronised flashing of fireflies [46]. The neuron has a quantity called activation or phase (which can be compared to the membrane potential of the neuron), which keeps on increasing, as the neuron 'spins', at a constant rate which can be considered as the angular velocity of spin. When the neuron gets an input spike, its activation rises by an amount proportional to the connection weight of the input. In-between input spikes, the neural activation spontaneously increases linearly as per the rate of spin or angular velocity. When the neural activation reaches a threshold value, it fires an output spike and the activation gets reset to 0 again, just like the phase of a spinning wheel gets reset to 0 as soon as it completes a rotation of 360°. In an activation-time graph of the neuron, the activation has a default slope and will therefore hit the threshold eventually even if the neuron gets no input spikes (which is similar to the behaviour of biological neurons, which fire spikes randomly at a low rate even if they get no input spikes). If the threshold value is Θ and the default slope of the neuron is m, the time when it reaches the threshold will be $t = t_0 + \Theta/m$ where t_0 is the current time.

The dynamics of this model are linear, and can be described as follows: Input spikes to the wheel model give a phase shift to the neuron, increasing its activation or phase by an amount equal to the product of the connection strength and the input sensitivity or significance of the neuron α. If the input connection weight

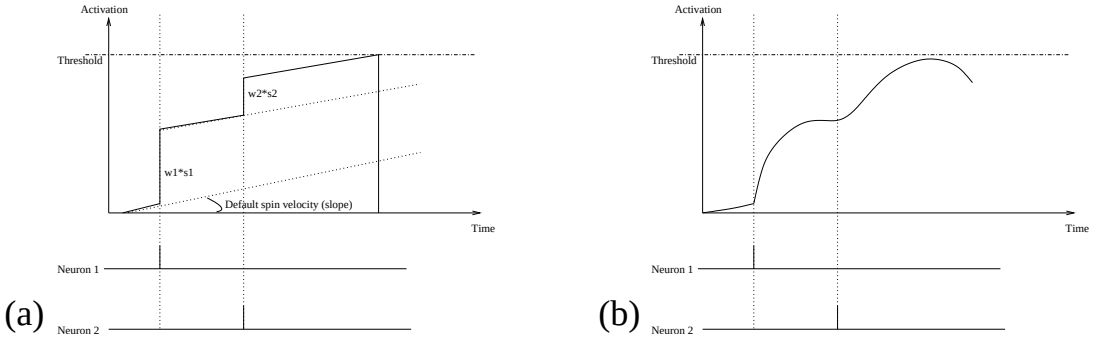

Figure 6.3: (a) The activation increase with respect to time in a spiking neuron following the wheel model. Once the neuron is activated, the activation normally increases at a constant slope. The neuron has two input spikes from Neuron 1 and Neuron 2 which cause its activation to jump by an amount corresponding to the product of the connection weight w and the sensitivity or significance s. (b) The equivalent plot in the RDLIF model for comparison.

between the j^{th} input neuron and the given i^{th} neuron is w_{ij}, the input sensitivity or significance of this i^{th} neuron when the j^{th} input fired was α_j, and the input neuron fires at time t_j then the activation of the i^{th} neuron at time t (assuming the neuron was activated at time t_0 , meaning that its activation slope started increasing from 0, say after a reset, at this time) is given by:

$$a_i(t) = m(t - t_0) + \Sigma_j w_{ij}\alpha_j \tag{6.10}$$

Now, this activation will reach the threshold Θ at time T (say).

$$a_i(T) = \Theta \tag{6.11}$$

$$\Theta = m(T - t_0) + \Sigma_j w_{ij}\alpha_j \tag{6.12}$$

Solving for T, we get:

$$T = \frac{\Theta - \Sigma_j w_{ij}\alpha_j}{m} + t_0 \tag{6.13}$$

Figure 6.3(a) gives an illustration of how the activation of the wheel model increases in response to successive inputs, until it reaches the threshold. Part (b) of the same figure shows how an RDLIF model with decaying activation and decaying activation rate would behave. As we can see, the models have some similarity in the shape of activation, although the wheel model is linear and the

RDLIF model is not.

6.3.1 Biological basis for the wheel model

The wheel model is a simple linear model and biological neurons generally have more complex dynamics. However, similar models have been used by modellers of the dynamical behaviour of complex systems, both biological and non-biological, such as synchronisation of flashing in fireflies [46] and desynchronisation in pacemakers [24], and communication networks such as sensor networks and pulse coupled oscillators (research in this is pursued by the DESYNC subgroup of Self Organising Systems group at Harvard University, among others).

6.3.2 Comparison of the wheel model with RDLIF model

A version of the wheel model (in which the default rate of activation increase is very low) can be considered a special case of the RDLIF model, if we assume (for the RDLIF model) the activation time constant τ_a to be very large (i.e. the rate of decay of activation is negligible) and the rate time constant τ_r to be very small (i.e. the rate is like an impulse function, it stays for a short time and then disappears, so the input spike in effect increases the activation rather than the rate).

In such a case, with an RDLIF neuron, assuming r_0 and a_0 in the RDLIF equations to be 0 the expression for the activation rate r, which is $\dot{r} = \sum_j w_j x_j - r/\tau_r$ where x is an impulse function representing input spikes, can be considered as an impulse function itself (since τ_r is small, the value of r decays almost instantaneously after giving a phase shift or jump to the activation a corresponding to the weighted input $w_j x_j$) giving a phase shift to the input as in the wheel model. The RDLIF expression for neuron activation $\dot{a} = r - a/\tau_a$ becomes on integration $a = w_j \delta(t)$ because the rate r decays almost instantaneously and $1/\tau_a \approx 0$. This case (activation a is constant except for rises due to impulses proportional to the input connection weight) is similar to the activation for the wheel model when the default slope m is 0.

6.3.3 Suitability of the wheel model for implementing the time-abstracted model

As long as the threshold is large enough so that the neurons strike the threshold on the slope rather than on receiving the input, i.e. assuming that all the neurons in a layer get their input spikes before reaching the threshold, and that all the neurons are in phase initially, we can guarantee that the Wheel model will give an output order that is identical to what we would get from the time-abstracted model. This is because the increase of activations in the wheel model, in response to an input, is identical to the corresponding increase in the time-abstracted model. The slope of the neuron only serves as a way to add time to the system without affecting the equation of the increase of activation in response to input spikes.

However, for this to happen, we need to ensure that all neurons in a layer are in phase when the layer gets the first spike from a burst. We do this by having an 'active' and 'inactive' mode for the wheel neurons, such that the phase or activation starts rising as per the default spin or slope only when the neuron is 'active'. We can assume all neurons in a layer to be in the 'inactive' mode initially, and program the first spike of an input burst to set all the neurons to the 'active' mode. When all the neurons in that layer have finished firing (i.e. when the N-of-M counter neuron on the output of the layer fires to reset all the neurons in the layer) the status of the whole layer is again set to 'inactive' until the next burst. We can justify this global status variable by assuming a weak connection from the input to all the neurons in the layer, such that all the neurons 'know' immediately when the first input spike fires (even those neurons with which the input neuron is not connected) and can change their mode to 'active' with the first input firing.

One problem with this model is the property that once activated, every single neuron has to fire at some time even if it gets no input spike, since the default rate of spin will ensure that the activation will equal the threshold eventually. We get round the problem by having a very low default slope (or rate of spin) relative to the threshold, and resetting all the neurons in the layer by using feedback reset inhibition, when the requisite number of them have fired (following the ordered N-of-M code).

An added property of the wheel model is that there are no explicit refractory periods (the period after firing a spike, when the firing neuron does not increase

its activation above the reset value) or latency between input and output. It is possible that the firing of an input spike will take the activation of the neuron above its threshold, which can create problems in maintaining the relative order of spikes in a burst. However, we can engineer the model to ensure that such a case does not happen.

In the remainder of this chapter, we will switch back to using the RDLIF model in some simulations on the stability of spike bursts, as the RDLIF model is more general, more accurately models a biological neuron and is similar to many standard spiking neural models such as leaky integrate and fire and spike response model. However, we will use the wheel model to simulate the complete spiking machine in the next chapter, as it is easier to tune it to behave precisely as we need.

6.4 Simulation of a spiking neural system to study stability issues

In this section we will examine, primarily through simulation, issues concerning modelling with spiking neurons, especially the stability of a burst of spikes in a feed-forward system. Our objective is to study the dynamics of the spike burst when it is propagated across many layers, and show that it is feasible to have stable burst propagation.

We simulated a feed-forward network consisting of partially connected neural layers of RDLIF neurons (similar to a synfire chain [1, 5, 40]). We fed the first layer a uniformly distributed random set of spikes, then fed the output spikes from the first layer into the second layer, the second layer spikes into the third layer and so on. There are no delays in the connection wires, apart from the delays inherent in the neural model. We then measured and plotted the temporal separation of the spike burst when passing through different layers. The architecture of the system is shown in figure 6.4.

6.4.1 Simulation method

For the simulation, we have a loop to model an array of feed-forward layers. Each iteration of the loop represents a propagation of a spike burst from layer i to layer (i+1). There is a different weight matrix in each iteration, representing different

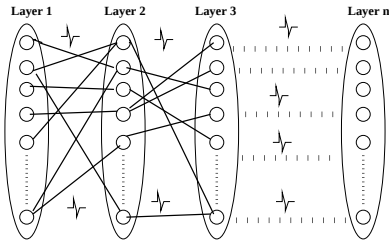

Figure 6.4: Architecture of the simulated network of a number of layers of 256 identical RDLIF neurons each. The neurons in each layer are connected to those in the next layer with 10% connectivity and random excitatory weights. The first layer fires a burst of spikes, which is fed to the second layer, whose outputs are fed to the third layer and so on. The temporal widths of the output spike bursts are measured.

neural layers but with the same average connectivity. The first layer is given a random set of spike firing times. We have an inner loop to count the time in time-steps in each such iteration, and in each time-step we check if any input or output neuron has fired. Each input spike increases the gain of the connected neuron proportional to the connection, as per the RDLIF model. The firings in the i^{th} layer cause spikes to fire in the $(i+1)^{th}$ layer. We wait for a specific time which is sufficient for all the neurons to fire. This time is chosen such that above this time, the gain and activation of all the neurons in a layer would decay and there would not be sufficient stimulus for any neuron to fire. The input and output spike firings occur simultaneously over this time period. We argue that this method (using time-steps and waiting for a specified time in each layer before moving on to the next layer) is equivalent to a pure event-driven system.

The process of propagating the spike firings from one layer on to the next is repeated for the next iteration after copying the output vector of spike timings to the input layer, and so on. Thus, in each iteration, the input is simply a vector of firing times and we get an output vector of firing times. We measure the temporal dispersion by taking the difference between the first and last neuron firing times in that burst. We then repeated the experiment for different values of input spreads, different values of network parameters, etc.

6.4.2 Sustaining stable activity in a population of neurons

One important issue in modelling a system of spiking neurons is sustaining stable activity over many layers, and preventing the spike burst from blowing up or dying

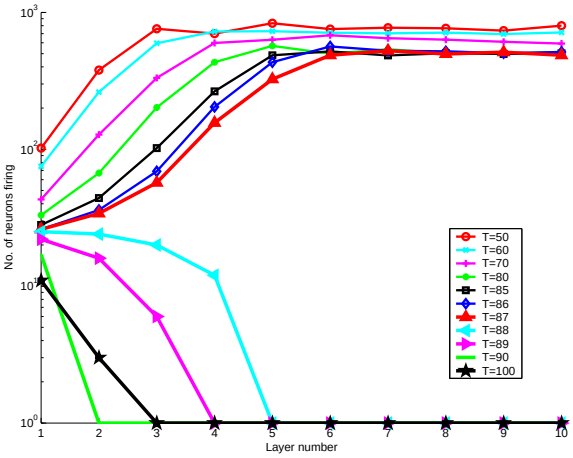

Figure 6.5: Plot of the variation of the number of neurons firing in each layer with varying threshold values, in a feed-forward network of 11 layers (including the input layer) of 256 RDLIF neurons each following 11-of-256 code, with 10% connectivity in each layer. The neurons have activation and activation rate time constants of 1 s each. The system behaviour switches abruptly from spiking activity increasing with each successive layer to dying out, as the threshold is progressively increased. The switch occurs at the threshold value of 87.

out as it passes through the layers. In our simulation, we have the same problem. The stability of the burst has some connection with the neural threshold, as modulating the threshold leads to change in behaviour of the spike burst. If the threshold is given an abnormally high value, the spiking activity dies out within a few layers, while if it is too low, the activity may increase with each layer to an unacceptably high level. We found that in our simulation, for a given network, there was no threshold such that the system could sustain a desired average level of spiking activity over infinitely many layers. The behaviour of the network abruptly switched from dying out of the spiking activity to an increase in activity with each successive layer as the threshold was progressively decreased, as shown in figure 6.5, with the switch occuring at a threshold of 87. It is interesting to note that the information content of a rank ordered 11-of-256 code is 87.6857 bits (as per the formula mentioned in section 4.1.1) which is also between 87 and 88, but this is just a coincidence, since the actual switch occurs between threshold values 87.693 and 87.694.

One of the ways of getting over this problem is through the use of feedback reset inhibition to control output activity. We keep the threshold in each layer such that activity at the output layer is slightly higher than the activity at the

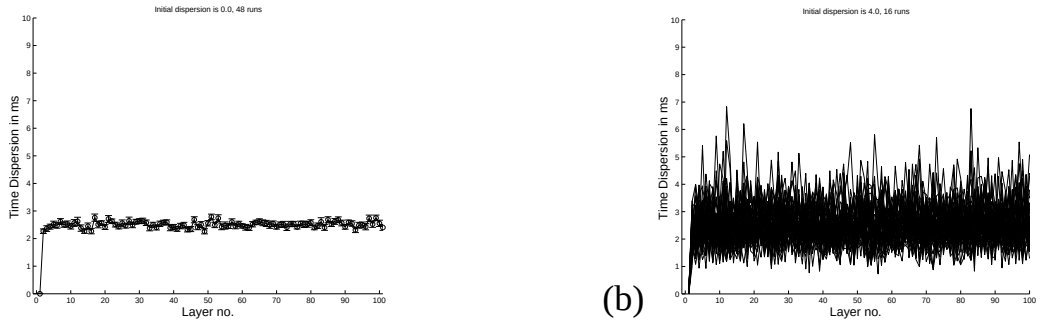

Figure 6.6: Plot of the average output dispersion, with initial input dispersion 0, of a burst of spikes passing through the 100-layer network, (a) averaged over 48 runs (b) plotted without taking averages

input, then we limit the activity by using reset inhibition and feed the spikes to the next layer. This can enable us to have a stable system in which a desired range of activity of firings could be sustained over a number of layers.

6.4.3 Effect of feedback reset inhibition

Feedback inhibition can be used to control the output activity of a layer, by suppressing output activity once a certain number of outputs have fired. As mentioned above, such feedback reset inhibition can be implemented by a neuron that is fed the output spikes from a layer, and fires an inhibitory spike once it gets a certain number of input spikes. This strong inhibitory spike resets all of the neurons in the layer. An RDLIF neuron with a threshold equal to the maximum number of spikes per burst (according to the code used) is equivalent to such a counter, provided the activation rate time constant is small and activation time constant large with respect to the input dispersion. This is what we described earlier as the method to implement N-of-M codes on the output of any neural layer.

In our simulation, we implemented an 11-of-256 code by using a counter to count spikes and resetting all neurons in the layer once 11 spikes have fired.

6.4.4 Simulation results

The parameters in our system are the individual neuron parameters (time constants, threshold) and system parameters (input time spreads and connectivity). Our base model has 100 layers with 10% connectivity from layer to layer. The

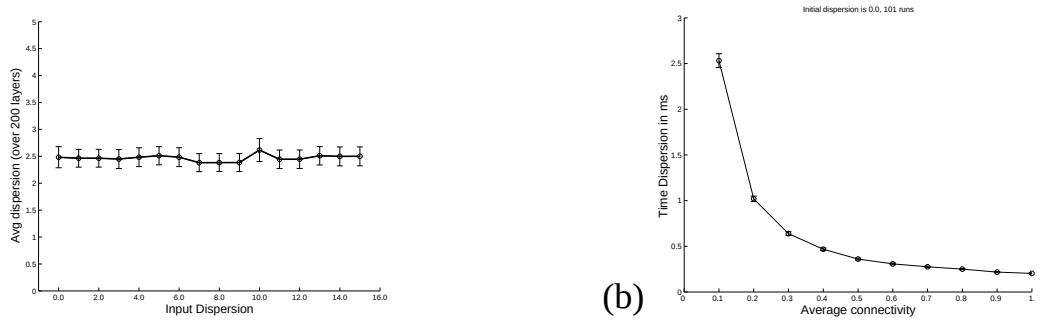

Figure 6.7: (a) Plot of the average output dispersion over 200 layers with varying input dispersions of a burst of spikes. (b) Plot of the variation of average output dispersion over 100 layers, averaged over 100 runs, with average network connectivity varying from 0.1 (sparsely connected network) to 1.0 (fully connected). As the connectivity increases, the dispersion decreases.

weights are real values between 0 and 0.1, chosen from a uniform distribution. The neural parameters are: threshold = 50, reset value = -1, time-step width = 10 ms, activation time constant = 1 second, activation rate time constant = 1 second. Each layer has 256 RDLIF neurons, out of which 11 are allowed to fire before the burst is propagated on to the next layer.

Figure 6.6(a) shows the average output dispersion (averaged over 48 runs) against layer when the initial temporal dispersion was 0. The plot is for a network of 100 layers, with average connectivity of 0.1 (which means that only 10% of the connections between successive layers are non-zero). Figure 6.6(b) shows the same plot without taking averages. We find that the temporal dispersion quickly tends to settle down into a narrow range, from its input value of 0, and does not vary much. As we can see, there is a value which is common to all the error bars. So we can claim that the spike temporal dispersion is within a controlled range. We repeated this experiment with different values of initial input dispersion and found that this range does not depend on the initial input dispersion. Thus, it is quite a stable and robust system.

Fig 6.7(a) shows the variation of the average output dispersion (averaged over all the layers) with the initial dispersion we gave to the first layer. We see that there is no appreciable change in the average dispersion, regardless of the input dispersion value.

6.4.5 Analysing the effect of network connectivity on temporal dispersion of spikes

In more realistic networks, it does not usually happen that all the layers are of similar dimensions or have the same average connectivity. Another interesting experiment is to investigate what happens to the temporal dispersion when we vary the average network connectivity. Figure 6.7(b) shows the average dispersion against network connectivity. We see a very interesting result: the average dispersion curve has a shape that is approximately inversely proportional to the connectivity of the network. When the connectivity is low, the neurons have difficulty reaching the threshold and so the average time spread or dispersion is higher. When the connectivity becomes higher, the average time spread of a burst becomes lower because the neurons can reach the threshold faster. Knowing this, in our model we set the parameters of different layers such that the average temporal dispersion of the spike bursts remains comparable across layers of different connectivity, which helps in making the system stable. We synchronise different bursts by 'padding' them with delays. Also, keeping the threshold higher for a specific layer increases the time it takes for the neurons in the layer to fire spikes.

We will analyse these issues in the specific context of the sequence machine in the following chapter.

6.5 Conclusion

From the simulations we can conclude that it is indeed feasible to have a system with stable dynamics of the burst propagation (input and output time dispersion) in a network of many feed-forward layers. The method to achieve these stable dynamics is to have feedback reset inhibition to control the spiking activity to a manageable level.

There are two important system-level time constants in our model, one for the temporal separation within a burst (our waiting time in each iteration from layer i to i+1) and the other for the separation between bursts. Since the average burst dispersion time in our experiments stayed within a defined range over many feed-forward layers, we could add external delays in the network to ensure that the inter-burst time interval is kept much larger than the burst dispersion time, so that successive waves of spikes do not impinge on each other. This would help to

ensure that the bursts are well-defined. We could also modulate the delays to help synchronise two different bursts, such as those in the input of the context layer in the sequence machine model. On the basis of this, we argue that a rank order code can be transmitted reliably and stably using a system of spiking neurons with feedforward and feedback reset inhibition.

In the next chapter, we shall consider the sequence machine system as a whole and implement a sequence machine using the wheel model of spiking neurons that behaves exactly like a time-abstracted system.

Chapter 7

A spiking sequence machine

In chapter 5 we described the model of a sequence machine consisting of an associative memory (a Sparse Distributed Memory using rank-ordered N-of-M codes) with a separate mechanism (a combination of a shift register and a context neural layer) to store the context or history of the symbol in the sequence. In this chapter we implement the complete sequence machine using spiking neurons. In the previous chapter we discussed various issues regarding implementation with spiking neurons, and showed how it is feasible to propagate a coherent and stable burst of spikes across many neural layers in a feed-forward manner. Here we use the wheel model of spiking neuron as described in the previous chapter, and implement a sequence machine that performs exactly as the system using time abstracted vectors.

The rest of this chapter is arranged in the following way: first we summarise the working of the sequence machine in terms of bursts of spikes. Then we indicate how to implement each component using spiking neurons, and how to make sure that the timing dependencies between different components are followed. After that we describe a spiking neural simulator suitably modified to accommodate some features specific to the sequence machine, and a method to visualise spikes emitted by different neural layers in the system. Finally, we use the simulator to learn a sequence with repeated characters, visualise the simulator output and verify that it is same as an equivalent system using significance vectors.

7.1 Functioning of the system implemented using spiking neurons

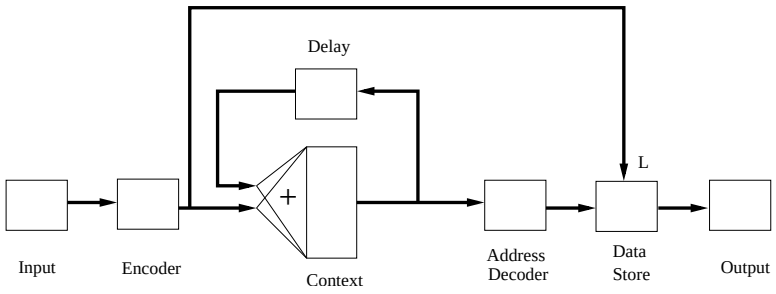

Figure 7.1: The complete network implementing the sequence machine

In chapter 5, we mentioned the structure of the sequence machine and the working of its various components, including the order in which the output vectors from one layer go to the inputs of another. In that description, we assumed the output ordered N-of-M vectors from different layers to be transmitted neatly from one layer to another, without any conflicts.

In the system implemented using asynchronous spiking neurons, each layer of neurons implements a component of the sequence machine, and interacts with other components. While implementing the system, we have to make sure that the firing of the bursts follows exactly the order of vectors in the time abstracted model and that the interaction between different components is smooth in order for the complete system to function correctly.

Figure 7.1 gives the structure of the complete sequence machine network, showing the various component neural layers (input, encoder, context, delay, address decoder, data store, output) and the connections between them. Below we will summarise the functioning of different layers of the system.

1. First of all, **a single spike is fed into the neuron corresponding to the input symbol** (thus following a 1-of-A code, A being the size of the alphabet). For example, if the input symbol is 'D' from the English alphabet and the input layer follows a 1-of-26 code, the fourth neuron gets a single input spike.

2. **The input spike is fed to the encoder layer**, which outputs a burst of N spikes corresponding to an ordered N-of-M code. These are fed to two

places which are connected to the encoder: the data store neurons (through the data inputs) and the context layer.

3. (The first of three cyclic steps of the sequence memory, as mentioned in chapter 5) **The encoder outputs are fed to the data store layer through the data inputs.** Initially, all the input connection weights of the data store layer of the SDM (with the address decoder layer) are zero. With the arrival of the first learning spike, the data store layer goes into the learning mode, which enables it to set the connection weights between the address decoder and data store layers of the SDM memory. However, since there is no stored order of the address decoder outputs from the previous wave in the synapses of the data store neurons, no learning takes place and the connection weights remain zero.

4. (The second of three cyclic steps of the sequence memory) Simultaneously with step 3, **the encoder outputs also feed to the context layer**. The context layer gets inputs from two layers, the encoder and the delay, but at this point the delay layer neurons (which feed back the old context from the previous wave) do not fire. Therefore the new context output is created by the context layer based on the encoded input only. The context outputs are computed following the combined model of encoding the context. The outputs of the context layer feed to the address decoder and the delay layer.

5. **The address decoder receives spikes from the context layer**, which enable it to produce its own output burst of spikes. The outputs of the address decoder are input to the normal (non-learning) inputs of the data store layer.

6. Simultaneously with step 5, **the context layer outputs also go to the inputs of the delay layer.** The delay layer has been explicitly programmed to have a high latency, so the output spikes from the delay layer, which are fed back to the inputs of the context layer, are not emitted until the next wave of firings from the input layer are started.

7. (The third of three cyclic steps of the sequence memory) **The outputs of the address decoder are fed to the data store neurons**, each of which stores the order of spikes in its synapses, in preparation for setting the weights. The weights are set when the data store gets data inputs from the

encoder neurons during the next wave. At this point the connection weights of the data store are still zero, as they were when initialised. Therefore, no data store neuron are expected to fire on receiving input spikes from the address decoder neurons. However, since the data store neurons follow the wheel model as described in chapter 6, they eventually fire when the spontaneous activation of the wheel neurons crosses the threshold. The feedback reset inhibition shuts off the data store layer after N of the M neurons have fired.

8. **The output spikes from the data store are passed through the output layer**, whose weights are the transpose of the encoder weights and which therefore acts as a decoder to translate the N-of-M code back to a 1-of-A code representing an output symbol from the given alphabet. Only one output neuron fires representing the output symbol, which is a prediction of the next input symbol. However, since no learning has taken place, this prediction is meaningless.

9. **The above cycle of steps (1-8) is repeated with each new input symbol.** The input spike, representing the start of the next wave, fires after the output spike from the last wave has fired (since the output is a prediction of the next input). The input spike is fed to the encoder layer, which fires N spikes. The encoder outputs and the delay outputs from the first wave (from step 6) are fed to the inputs of the context layer, which generates the new context as per the combined model. The encoder outputs also go to the data inputs of the data store, which set the connection weights between the data store and encoder to associate the data input spikes with the address decoder spikes from the previous burst (which it had stored in the synapses in step 7) as per the learning rule. The context outputs go into the address decoder, whose outputs go into the data store (whose synapses again store the order of significances for the next wave) and the data store neurons fire their output spikes. Finally the data store outputs go into the output layer which generates the prediction for the next input symbol.

Figure 7.2 illustrates the various processing steps in the sequence machine, as described above.

The above steps describe how we expect the system implemented in spiking neurons to behave. In the actual implementation, there are some issues (mainly

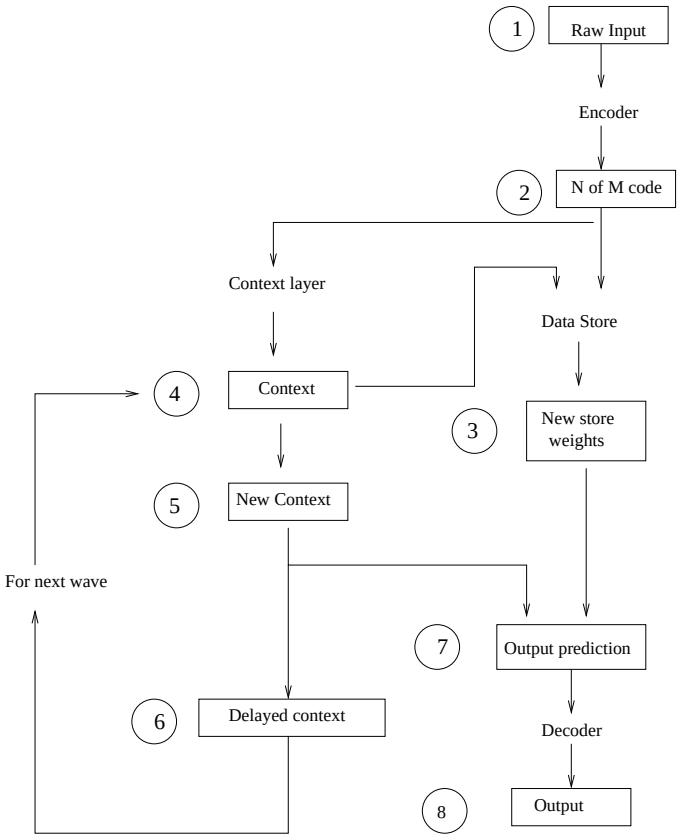

Figure 7.2: Flowchart of the information flow in the system, showing the sequential steps in time

regarding the timing relations) we need to resolve before the system can function in this precise order, some of which were mentioned in the previous chapter. In the following sections we will examine them in more detail.

7.2 Timing relationships between components

In implementing a system using spiking neurons, the only information conveyed is in the timing (or relative timing in the case of rank-order codes) of the spikes emitted by any layer. Therefore it is important to ensure that the relative timings of different layers of spikes are controlled precisely.

We define the latency of a given neural layer as the time between the middle of the input burst and the middle of the output burst (where the middle of a burst refers to the mean of the firing times of the burst).

7.2.1 Timing dependencies on signals external to the system

1. **The output spike must occur before the first spike of the next input burst**, since the output is the prediction of the next input. This could be accomplished if there is some kind of handshaking communication between the output and input layers, such that the input layer knows when the output layer spike has fired and it can fire the next input spike. However, for simplicity we will assume that the inputs are all equally spaced, and will engineer the time gap between successive input spikes in such a way that this constraint is met. Therefore, the temporal gap between inputs should be bigger than the time taken for a spike burst to propagate through all layers of the system in the forward direction (excluding the feedback through the delay layer).

2. The delay latency should be such that **the delay spikes input to the context layer from the last wave of spike firings arrive at approximately the same time as the encoder spikes from the current wave**. This means synchronising an internal variable (the delay latency) with a signal that is outside the system (the input spikes) in such a way that the encoder spikes fire at around the same time as the delay spikes.

We do this by engineering the delay layer latency to synchronise with the time gap between inputs.

The constraints mentioned above concern input timings, which are external to the system. However, inside the system, we assume that the input spikes coming in are already synchronised to the delay and outputs, so we will not worry about the problem.

7.2.2 Timing dependencies between spike bursts from various layers

1. In the learning mode, we have to make sure that the **store neurons receive all the spikes from data inputs (and complete writing the association by setting the connection weights) before receiving the spikes from the context** through the address decoder (which have to be stored until the next input wave arrives). We can assume that this condition is fulfilled because of the structure of the network, since the encoder spikes go directly to the data inputs of the data store, while the address decoders fire only when the encoder spikes have passed through two more layers, which are the context and the address decoder.

2. Initially the delay layer has no inputs (because there is no previous context) and consequently the context layer will fire more slowly (since its delay inputs are missing) than it would normally. We have to ensure that this does not destabilise the system, and **the inter-burst interval should stabilise after passing through a few layers**. However, from the previous chapter we saw that in a feed-forward system of neurons with a feedback inhibition mechanism, the time dispersion of a burst does stabilise when it passes through many layers. Therefore, we can assume that this condition would be met.

Figure 7.3 shows the connections between the major components of the system (input, context, delay and data store layers) and the timing dependencies between them.

In order to engineer a neuron to have a predetermined latency (the time lag between its input and output spikes), we can control the threshold to increase or decrease its latency. Increasing the threshold of a neuron will make it spike

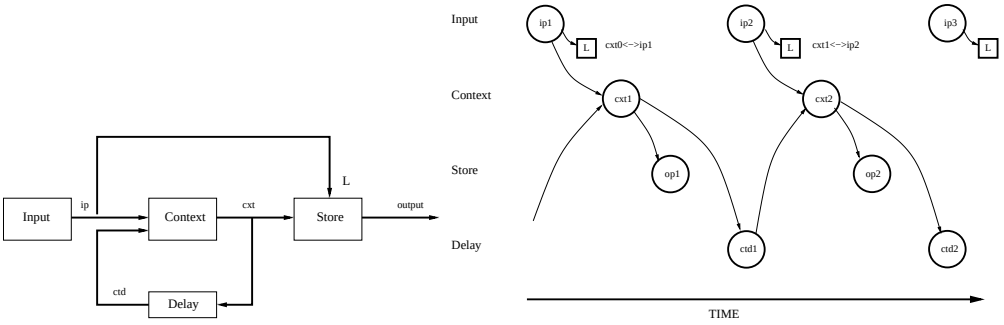

Figure 7.3: A graph showing the timing relation between a few major layers of the system, with time on the X-axis. Connections between these layers are shown on the left

later, thus slowing down the whole system. Decreasing the threshold will have the reverse effect. For the wheel model, another such control could be the default rate of spin, or slope of the activation: slower the spin rate, higher will be the latency.

In setting up the parameters related to the component timings, we have to observe the following conditions:

1. If we increase the time gap between successive inputs, the delay needs to be increased correspondingly (by increasing the threshold appropriately) to make sure that the two inputs to the context arrive at nearly the same time. A similar condition holds true if we wish to decrease the gap between successive inputs.

2. If we increase the delay layer latency, then in order to balance it, we will have to also increase the gap between successive inputs.

3. If we increase the latency of any layer between input and context (i.e. the input or encoder), we need to proportionately increase the next input firing time keeping input output latency constant (because firing times of all successive layers will be shifted by same amount)

4. If we increase the latency of any layer between context and output (i.e. the address decoder, data store or output) we need to proportionately increase the next input firing time, for which we also must increase the delay layer latency as mentioned earlier.

5. Layers which have more inputs are expected to fire quicker. In order to enhance the stability of the system, we want the latency across layers to be

comparable. Therefore, these layers should have their neural thresholds set to higher values. In our system, the context layer has inputs from both the encoder and delay layers, so its threshold is set higher.

In our simulation experiments with the spiking neural system, we have made sure that these conditions are met. The latency of a neural layer can be set to a higher or lower value where needed by tweaking the time constants of the spiking model. The timing of the input spikes to the system is within our control, so we have ensured in the simulation that the timing restrictions are followed and the spikes are input to the system with regular time intervals corresponding to the latency of the whole system (total latency of all the layers in the sequence machine).

7.3 Issues specific to the sequence machine

Apart from the timing issues mentioned above, there are some other issues specific to the sequence machine system, that we had not covered earlier. These issues are briefly summarised below.

1. We need to **store the rank-ordering of the two input bursts to the context from the delay and input layers separately**, since the increase of activation of the context on receiving any input (which is a function of the sensitivity of the layer, which decreases with each additional input spike in the burst) depends on the position of that input in its respective burst. This is done by having two different feed-forward shunt inhibition (FFSI) neurons on the two kinds of inputs to the context, which keep track of the rank of the input spikes from both the layers.

2. We have to **ensure that output spikes of any layer do not start firing before all the input spikes from the layer before it have arrived**, else it will spoil the code being transmitted. This can be arranged by having large thresholds and low default rate of spin, along with extra axonal (between an input and output) or wire delays.

3. The system is sensitive to noise in the spike trains and a few random spikes have the potential to upset the outputs (mainly because it waits for N neurons in a layer out of M to fire until it resets a neural layer, in accordance

with the strict N-of-M code we are using), so it needs to be **carefully engineered so that the times of firing are synchronised**. In simulating the system, we have assumed that there is no noise in the system. However, since we are using ordered N-of-M code and the number of actual codes used in the alphabet is far less compared to the total possible number of codes, there is some degree of redundancy and therefore some error is tolerable.

For example, using an alphabet of 26 symbols taken from an ordered 11-of-256 code, the total number of possible ordered codes is $\frac{256!}{245!}$ which is to the order of 10^{26}, while the number of symbols used is only 26.

4. The layers have control over their output spikes, but have no knowledge of how many input spikes are going to arrive and when (unless each layer asks the layer before it). Therefore we have to **ensure that the output spikes follow the correct code and there are no errors in the generation of the output spikes**, else the neurons in the next layer could keep waiting indefinitely for the expected number of input spikes.

5. Initially, the context layer has no inputs from the delay layer. Since we are using a counter on the inputs to determine when to reset the inputs to a layer and also to determine the significance (just as we have another counter on the outputs to enforce the N-of-M code) and since the model cannot distinguish between different kinds of inputs unless we specifically program it to do so, we need to **initialise this counter to the number of delay layer outputs** (i.e. treating it as if the delay neurons have already fired).

6. In the data store memory, we write the connection weights to the memory based on the associations between two significance vectors: the vector corresponding to the spike burst from the encoder layer to the data inputs of the data store (this burst is current and so need not be stored), and the stored vector corresponding to the previous burst (during the last wave) from the from the address decoder layer to the normal inputs of the data store (which needs to be stored till the next wave). Therefore, we have to **store the significance vector from the last wave in the synapses of the data store neurons**, until the data inputs come and learning takes place. After the learning is completed (signalled by the N-of-M counter

neuron at the inputs firing to indicate it has received all the expected N input spikes in the current burst) the stored synapses are reset in preparation for the next burst.

7.4 Implementation of the individual components of the system

Below are some added observations on the implementation of some individual components of the system using spiking neurons.

7.4.1 Implementing the context layer

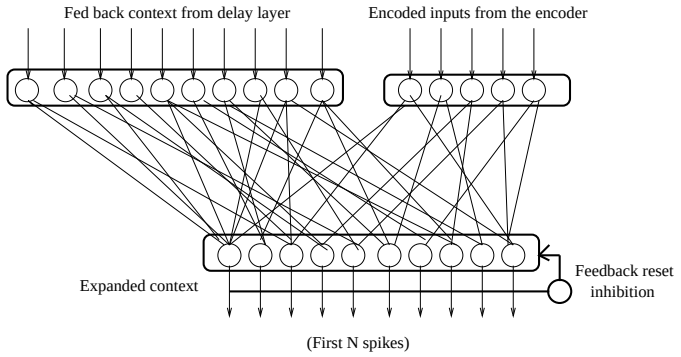

Figure 7.4: Formation of the new context from the input and old context, implemented in neurons

'Addition' of two bursts of spikes may be implemented in neurons by having both the bursts (whose corresponding significance vectors are to be added) input to a neural layer at the same time. The output of such a layer will effectively represent the sum of the bursts. In the combined model, the context is modified in the following way: the fed back previous context spikes from the delay layer are fed to the context neural layer in order to project the context into higher-dimensional space. Spikes from the encoder layer, representing the encoded present input, are also fed to the context layer. The effect of the spike bursts from the encoder and delay layers on the activations of the context neurons is additive. N output spikes are selected from the context layer in order by means of feed-back reset inhibition (to implement the ordered N-of-M code on the new context).

The context sensitivity λ can be implemented by multiplying the weights of the delay layer (that feeds back the old context) by λ, so that the effect of an

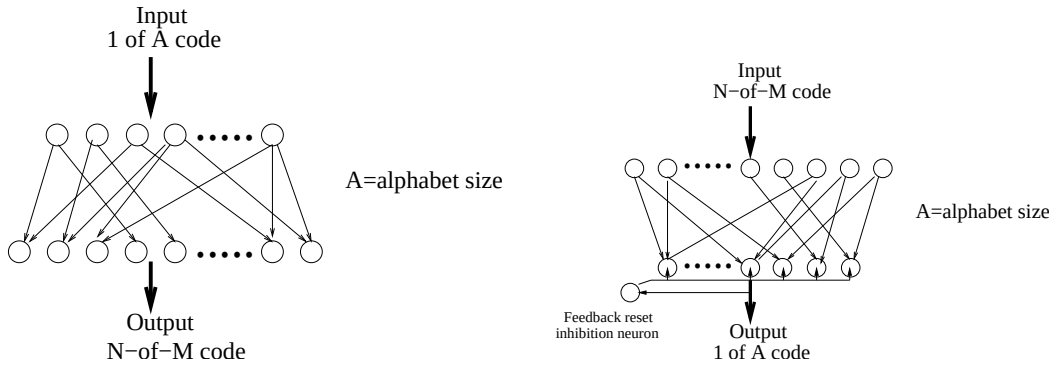

Figure 7.5: (a) Design of the encoder network to convert the input symbol (1-of-A code where A is the length of the alphabet) into an ordered N-of-M code. (b) The decoder network to convert the output of the memory (N-of-M code) back into a 1-of-A code (by means of feedback reset inhibition) and thus find the closest symbol match, which is the output prediction for the next symbol in the sequence

encoder layer spike on the activation of the context neuron is proportionately higher than the effect of a delay spike. Another way to implement the context sensitivity factor could be to weight the increase of activation of context neurons in response to delayed old context spikes by the value λ, thus ensuring that an encoder spike (representing the new input) has a proportionately higher effect on increasing the activation of a context neuron than a delay spike (representing the old context).

7.4.2 Encoder and decoder

The purpose of the encoder is to translate a 1-of-A code (corresponding to an input symbol in the alphabet) into an ordered N-of-M code. Our encoder is a neural layer with fixed weights (no learning of the weights is involved) and therefore functions as a lookup table. The purpose of the decoder is to convert the data memory (or data store) outputs (which is an ordered N-of-M code) back to a 1-of-A code, representing one of the symbols in the alphabet which is the prediction of the next input symbol. Figure 7.5 shows the structure of the encoder and decoder layers when implemented using spiking neurons.

The decoder layer has an ordered N-of-M code as input and outputs a single spike corresponding to a 1-of-A code. By a 1-of-A code we mean that only one neuron fires a spike at the output layer, which is the symbol in the alphabet corresponding to the identity of the neuron, in the same way as the input layer.

The decoder is layer is implemented by having a feedback reset inhibition neuron that cuts off the output activity after the first neuron in the layer has fired. The decoder weights are the transpose of the encoder weights, so in effect it performs a reverse lookup of the code.

7.4.3 The data store layer and the learning rule

The data store layer is the only component in the sequence machine where learning takes place. The other layers input, accumulate and output spikes by performing calculations involving significance vectors, but do not have any long term learning component involved in them. The layer stores associations between significance vectors corresponding to the spike bursts in its inputs from the encoder and address decoder. We have assumed throughout that the spiking neurons have a localised learning mechanism (all information needed to decide when to spike is stored locally, either through the input spikes or stored in the synapses) and fire asynchronously (there is no global clock synchronising the whole system, timing each component precisely).

The learning rule for setting the weights is summarised below:

$$w_{ij}(new) = max[w_{ij}(old), \sigma_i \sigma_j^{'}] \tag{7.1}$$

where w_{ij} = weight component between the i^{th} data store neuron and the j^{th} address decoder.

$\sigma_i = i^{th}$ component of the significance vector corresponding to the spike burst from the encoder layer, which form the data inputs of the address decoder.

$\sigma_j^{'} = j^{th}$ component of the significance vector corresponding to the spike burst from the address decoder layer.

What this formula means is that the new connection (weight) between the i^{th} address decoder and the j^{th} data store neuron is set to the maximum of the old weight and the product of the significance vector components of the i^{th} address decoder neuron and the j^{th} data input neuron.

Although the max operator is strictly not biologically plausible, it can be understood as a kind of dynamic threshold set to the old value of the connection weight, such that if the product of significance vector components of the address decoder and data store exceed this threshold, the connection weight is set to the new value.

The weight change algorithm requires each data store neuron to store the significances of the data inputs from the encoders and the normal inputs from the address decoders. However, this depends on the order in which its two types of inputs fire. If the data inputs had fired before the address decoders, we would have to simply store the data input significances and when each address decoder fires, the weights are set at that moment to the maximum of the old weight and product of significances. However, in our case, we have the address decoder neurons firing first. This presents a problem of storing all the significances of address decoder outputs in the data store neurons. **The only place we can store the significances until the next input wave are the synapses of the neurons, therefore each synapse must have a memory to store the input sensitivity**.

Another issue we need to consider is: **how does the data store layer know when to switch modes from learning (setting the connection weights) to recall** (calculating the outputs)? For this purpose, we stipulate that the learning mode is triggered on the firing of the first encoder spike on the data inputs of the data store, and is finished when the last encoder spike has fired (when the feed back reset inhibition neurons on the output of the encoder implementing the N-of-M code fires, signalling that the last encoder spike has fired in the present burst).

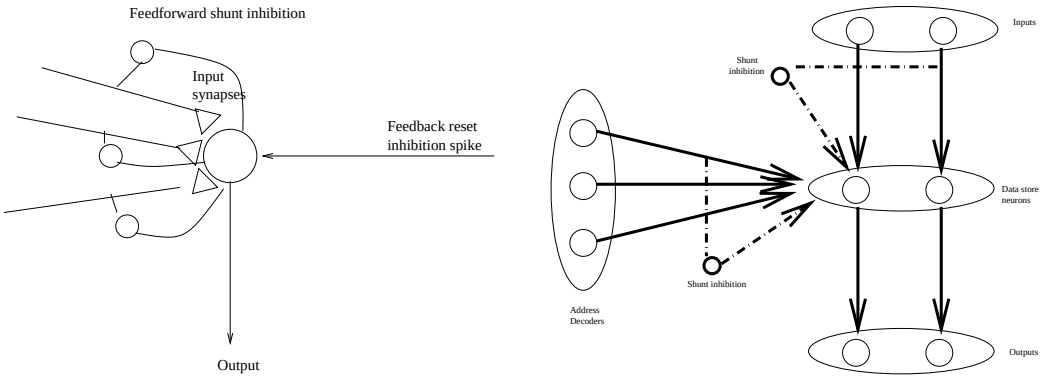

Figure 7.6: (a) A data store neuron (b) The data store memory implemented using spiking neurons

Figure 7.6 shows how the data store layer can be implemented using spiking neurons.

The learning rule we implemented using spiking neurons was described in chapter 4. This is not a standard algorithm for spiking neurons, but

this is the one that is used in the SDM memory [27, 26] and is also based on a localised Hebbian-style learning rule. This is applicable in the case when the two inputs to be associated arrive at different times (as is the case in the sequence machine), so there has to be some kind of learning eligibility, akin to a standard eligibility trace mechanism in neural systems and classical conditioning [79, 19, 86], that is stored when the first input has passed away, which we have implemented by the significances stored in the synapses. This is the algorithm we have implemented using spiking neurons.

We postulate that the **learning sensitivity, or eligibility trace of the neuron stays constant unless changed externally**. This is the quantity that is stored in the neural synapses of the data store layers when the first wave of spikes has passed. During the first wave, the address decoder spike burst arrives at the data store, and the neurons store the significance vectors or eligibility traces corresponding to this burst in their synapses. When the data inputs arrive during the next burst, the weight change takes place as per the learning rule, and the stored eligibility traces are cleared.

The k^{th} data store neuron has one feed forward shunt inhibition or FFSI neuron (and one significance or sensitivity value) corresponding to the data input neurons, which takes input spikes from all the data input neurons and outputs an inhibitory spike to the k^{th} data store neuron every time one of the data input neurons fire. The effect of this inhibitory spike is to turn down the sensitivity corresponding to data inputs by the significance ratio α. This neuron also has another FFSI neuron (and another significance or sensitivity value) corresponding to the address decoder neurons, which takes input spikes from all the address decoder neurons and outputs an inhibitory spike to the k^{th} data store neuron every time one of the address decoders fire. The effect of this inhibitory spike is to turn down the sensitivity corresponding to data inputs by a factor, say β.

Apart from this special mechanism of feed forward shunt inhibitions, the rest is simply an implementation of **localised Hebbian learning** (as in the unordered case as described in Furber et al [27]): the weight component (i,j) between the i^{th} address decoder and the j^{th} data store neuron is changed only when both the i^{th} address decoder and the j^{th} data input neuron have fired (in some fixed order, say address decoders before the data inputs). Also, the neuron needs to determine when both its data input and address decoder have fired. This weight component is set to the maximum of the old value of the weight component and the product

of the address decoder and data input sensitivities (which are both stored in the j^{th} data store neuron by the mechanism that is described above).

Finally, there is another small issue: **the significances of all the neurons in the layer should be reset when the firing burst is set**, in order to enable it to start storing the next burst of sensitivities. This will need yet another kind of feedback reset inhibition, which will be triggered in the appropriate case. The first feedback reset inhibition is triggered when it has counted N spikes, and fires a resetting spike to shut off the whole layer.

7.5 Implementation of the simulation

There are quite a few neural simulators available, such as GENESIS, NEURON, SNNS (Stuttgart neural network simulator), Matlab neural network toolbox, SpikeNNS, etc. They have varied functionality and complexity. For the purposes of implementing the sequence machine, we chose to use a locally designed spiking neural simulator called 'SpikeNetwork' designed by Mike Cumpstey, a researcher working in the APT group of Manchester University [17], that was quite suitable for our purposes and for the model described. The reason we used a locally built simulator rather than a more standard one was because we had to modify the simulator code to suit the application, and having a locally built simulator made the modification task easier.

7.5.1 Features of the simulator

The main features of the simulator are described below.

- The simulator has **reusable components**, both low-level and high-level. It can easily be configured to implement any kind of system, with any number of neural layers, connection structure, or neuron parameters, and so is highly flexible. All we need to do is to specify all the configuration information in a configuration file.

- It can work with a **wide variety of spiking neuron models** such as leaky integrate and fire, RDLIF model, Izhikevich model [36] and wheel model. It is relatively simple to implement new spiking neuron models if necessary, since the code is modular and object oriented.

- The simulator can be programmed to **process different model of neurons within the same network**, as specified in the configuration file.
- **Each neural layer can function in two modes: learning or recall**. The learning mode is triggered by special 'learning' inputs, although the simulator works in the recall mode by default.
- **Each layer can have more than one type of input**. In the sequence machine, only the data store layer goes into the learning mode and only the context layer has input spikes from more than one layer.

7.5.2 Working of the simulator

The simulator is given a configuration file for the network and another file specifying the timing of the input spikes. The system starts up with spikes from the input layer (with firing times and input configuration specified in the network configuration file and the simulation file) and gives as output a series of spikes with firing times. Each input spike sets off a wave of firings in different layers of the system.

We have developed a plotting program in Matlab, which then takes in the series of output spikes and plots them with annotations and labelling to show clearly the timing relations between spike bursts from different layers.

The format of the input configuration file, simulator output and other details are described in Appendix B.

7.5.3 Simulator features specific to the sequence machine model

A few changes had to be made in the design of the basic simulator to incorporate features specific to the sequence machine and make it work exactly as in the time abstracted model.

One problem that arises in the case of context layer is that it has two sets of input spikes coming in from the encoder and delay layers. In determining the respective significance values to calculate the increase in activation, we need to know the temporal rank-order of each input spike in its respective input burst. Therefore, we need to keep two separate counters for knowing the ranks of input spikes to the context layer coming from the encoder and delay layers.

A different vector is needed to store the learning significance vector in the synapses of each neuron, which is not reset until the next wave of spikes.

Since all inputs to the context layer have a unique ID and there is no explicit way to remember which of the input layers emitted the spike, the layer has to be deduced from the identity of the input spike, and after that the rank of inputs from that layer has to be incremented. Only then can the significance value and the increase in activation of the context neuron be computed. This illustrates a problem regarding passing of information across different classes in the object oriented simulator. In the neural layer class, we need to keep an array of learning significances, which we have to make sure is not reset when the rest of the layer is reset (i.e. when the output spikes of that layer have been fired, determined by the counter corresponding to the number of output spikes equals the maximum number as per the output N-of-M code of that layer). We need to do this because the setting of connection weights in the data store layer takes place when the current wave passes and the next bursts of inputs propagate through the system.

In order to make the system implemented using the wheel model work exactly as the system implemented using the significance vectors, we have to make sure that all neurons in the neural layer have their phase synchronised. Therefore, we 'activate' the layer with the first spike of every burst such that the neural activation starts rising only after such an activation signal is received.

Each layer in the system has the following counter neurons:

1. An **input counter neuron**, connected to all inputs to the layer, to count the rank of input spikes in order to compute the significance, which also activates all the neurons in the layer on receiving the first spike of any burst. This counter neuron is used for decreasing the input sensitivity or significance progressively by multiplying the present significance value for the layer by a constant significance factor α which is less than 1. Since the rank counted by this neuron does not need to be stored but is used up immediately to calculate the significance, having only one counter for the whole input layer is sufficient instead of a separate counter for each neuron in the layer.

2. A **second input counter neuron**, in case of layers with two types of inputs such as the context layer, which counts the rank of the input spike in the burst coming from the second input layer, and uses it to compute

the significance and increase of activation in the same way as for the first counter neuron.

3. A **learning counter neuron** for the data inputs (from the encoder) of the data store layer, to count the rank of the data spike and the learning sensitivity or eligibility, which is not reset when the whole layer is reset (when the resetting spike from the output counter neuron has fired). In case of learning the change of weights takes place when the data inputs strike the encoder, because the significances of the address decoder neurons from the last burst has already been stored in the synapses of the neurons. Therefore in this case too, only one learning counter neuron is needed for the whole layer (instead of a separate counter for each data input) because the significances are calculated instantly and used for changing the connection weight, and do not need to be stored. Whenever a normal input spike from the address decoder to the data store fires, no learning takes place, and when a data input spike from the encoder fires, change of the weights takes place. This is how the system automatically decides when to change the weights (in write mode) and when to simply increase the activations (in read mode).

4. An **output counter neuron**, which is connected to all the outputs of the layer and fires a resetting spike to reset the activations of all neurons in the layer and also deactivate the layer (so that the wheel neurons do not start spinning with the default rate, thus spontaneously increasing the activation, unless the layer is activated).

The N-of-M code is ensured for each layer by the input and output counter neurons, the input counters for enforcing rank-order sensitivity by decreasing the significance value multiplicatively for each successive input spike, which is used to calculate the rise in activation, and the output counters for ensuring that exactly N neurons fire on the output layer out of a total of M neurons, thus enforcing N-of-M coding.

7.5.4 Using time steps and event queues in simulation

There are two ways to go about the simulation of neural systems: using an event queue and using timesteps.

The timestep model

As the name indicates, the timestep model of simulation of a system involves having a global clock counter, and recomputing the state (with state variables such as activation, learning sensitivity or eligibility, input sensitivity, activation rate, etc) of every neuron in the system at each timestep. Implementing a simulator following this model is quite simple, but it is not efficient to recompute the state of the system at each timestep, especially when the system is large and the number of firings per timestep is small compared to the size of the system. Also, if the firing times are quite close to each other, the timestep has to be sufficiently small to fire them in the right order.

The event queue model

Another way to simulate a spiking system is to have continuous time instead of timesteps, and have an event queue. The firing of a spike at a specific time is defined as an event. An event stores two items of information: the identity of the neuron which is to fire, and the expected time of firing. The recomputing of the neurons takes place only when an event is fired. Events are stored into a queue for processing, which is kept sorted in order of firing times. The queue has two main operations:

1. **Insert** an event into the queue in its correct position such that the queue remains sorted in its order of firing times

2. **Remove** an event from the front of queue and process it. Processing a firing event means re-calculating the expected firing times of all the neurons connected to the firing neuron, because these are the neurons that receive the input spike, so their expected firing time will be closer to the initial value.

The queue is initialised by the firing of a spike from the input layer. It is operated by removing events from the top of the queue one by one and processing them, until the queue becomes empty.

In some neuron models, it is not possible to compute the exact time of firing accurately, because their kernel equations are not invertible. For such models we can still compute the expected firing time within an adequate degree of accuracy

by using numerical methods. This problem does not occur with the wheel model, which is linear.

The event queue model, although more complex to implement than the time step model of simulation, is more efficient in case of small networks with sparse firing activity, because there is no need to keep a global time counter or to recompute the state of every neuron in the system at each time step, but only when the spikes actually fire. However, a problem with using an event queue is that it can become quite long, and it may not be efficient to recompute firing times and delete and insert event nodes from a long queue. This also depends partly on which data structure (such as a sorted array or linked list) is used for the implementation of the queue. Also, when the expected time of firing is far from the current time, there is a possibility of having to perform this delete and insert operation a number of times before the firing event takes place.

The SpikeNetwork simulator is implemented using a time step model in combination with a small event queue for events whose firing time is in the near future.

7.6 Simulating the complete system

We simulated a sequence machine using the SpikeNetwork simulator, using the Wheel model of a spiking neuron and the context encoded using a combination of neural layer and shift register. The sequence machine had a structure as shown in figure 7.1, and all the assumptions and modifications discussed earlier were followed. Since we used the linear wheel model, we could calculate the approximate latency of each layer, and change the latency if necessary (by changing the threshold or the slope of the wheel model representing the spontaneous rate of increase of activation). Based on this, we designed the input timings and the delay latency so that the timing dependencies between various layers are maintained. The input and output layers follow a 1-of-20 code, the encoder and data store layers follow an ordered 3-of-20 code while the other layers follow an ordered 6-of-40 code, each with significance ratio α equal to 0.97. The context sensitivity factor λ is kept at 0.97 as well. The default wheel neurons were set to have an activation rate of 0.1 and threshold equal to 6.0, although these parameters were set differently for the context, delay and data store layers, in order to make their average latency approximately same as the other layers.

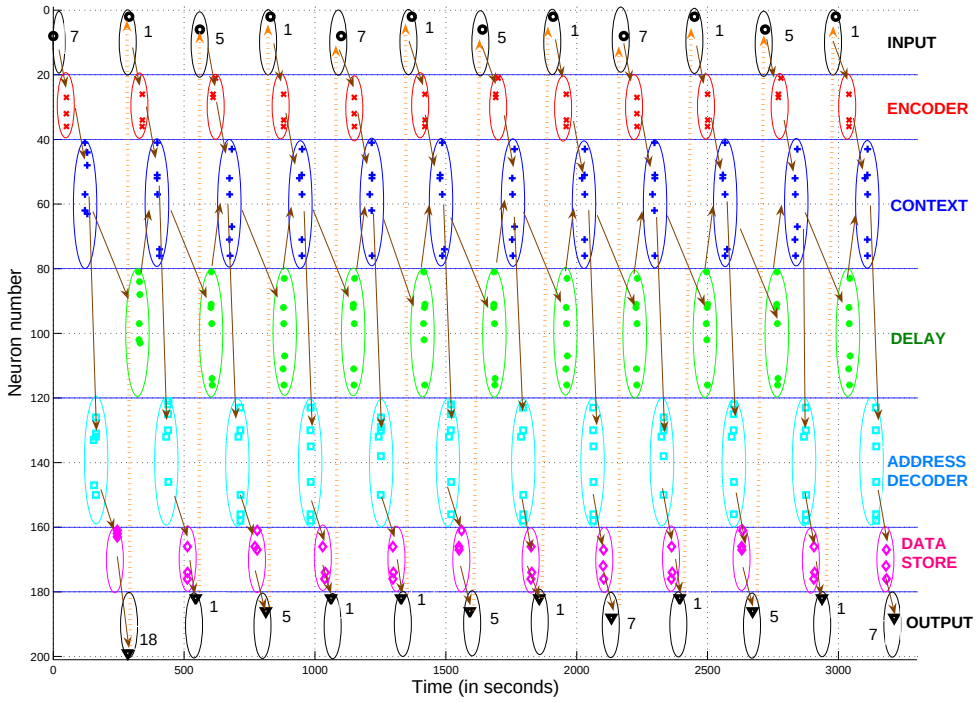

Figure 7.7: Plot of spikes emitted by different layers in the sequence machine against simulated time (actual time to run the complete simulation is about 4 minutes). Spikes of the same colour, encircled by coloured ellipses, belong to the same layer. The brown arrows denote causality, how a burst of spikes causes firing of another burst in the next layer after a delay, and the orange dotted line links the prediction on the output to the next input. The figure plots 12 different waves of spikes each triggered by an input spike, and forming the sequence 7,1,5,1,7,1,5,1,7,1,5,1. After the first 7,1,5,1 input (when the predicted output is 18,1,5,1 which is incorrect but the system learns the sequence), the predictions 1,5,1,7,1,5,1,7 of the next input symbols are correct. The input spikes are uniformly spaced in time.

Figure 7.7 shows the output of the simulator (with spike outputs of different neural layers across time) following the combined model, on being given a repeated input sequence '715171517151'. The first time the sequence '7151' is given the output prediction is incorrect, but the system learns to predict correctly and the next time the '7151' is entered the prediction is correct. The sequence '7151' is used because it is the simplest sequence where we need to have knowledge of context to determine the successor of the symbol '1' (since the symbol '1' is repeated with two different successors, the correct prediction of the

successor of '1' depends on its context).

In the diagram, spikes from different layers are plotted on the Y-axis and time on the X-axis. Spikes of the same colour and in the same horizontal band belong to the same layer (which is labelled on the right), and the arrows show how a burst of spikes from one layer causes the next layer to fire a burst after some time. We can see from the diagram that the spike bursts from different layers are coherent, stable, well behaved, and follow the timing dependencies mentioned. They also implement the high-level sequence machine by learning the given sequence 7151 in a single pass, and predict it correctly in the second and third presentation of 7151.

7.6.1 General observations from the sequence machine behaviour in the given example

From a detailed examination of the time and identity of spikes fired in different layers (we can get a rough idea from the diagram) we can make the following observations:

1. The prediction of the next input is correct the second time (and also the third time) the sequence 7151 is input after it learns them the first time, thus confirming that the sequence machine has successfully learnt to predict the sequence.

2. The input-output latency (time taken for a burst of spikes to propagate through the whole system from the input to the output layer) is higher during the first wave of spikes, because the delay inputs to the context do not fire (so it takes a bit longer for the context neurons to fire). However the latency stabilises in the next few waves.

3. In each layer the last input fires before the first output, fulfilling the condition that all inputs to a layer must fire before the first outputs of the layer start firing, and the time gap between the last input and first output stabilises in all the three waves (i.e. the inputs are not "catching up" on the outputs).

Thus we can conclude that the sequence machine based on the combined model is behaving stably for all the layers and is learning the input sequence correctly after a single presentation.

We then repeated the experiment using the neural model of context encoding instead of the combined model. Here too we used a similar network as the earlier experiment (but used four delay layers with latency, measured by the default slope of the wheel model, equal to all other layers, instead of one delay layer with a big latency).

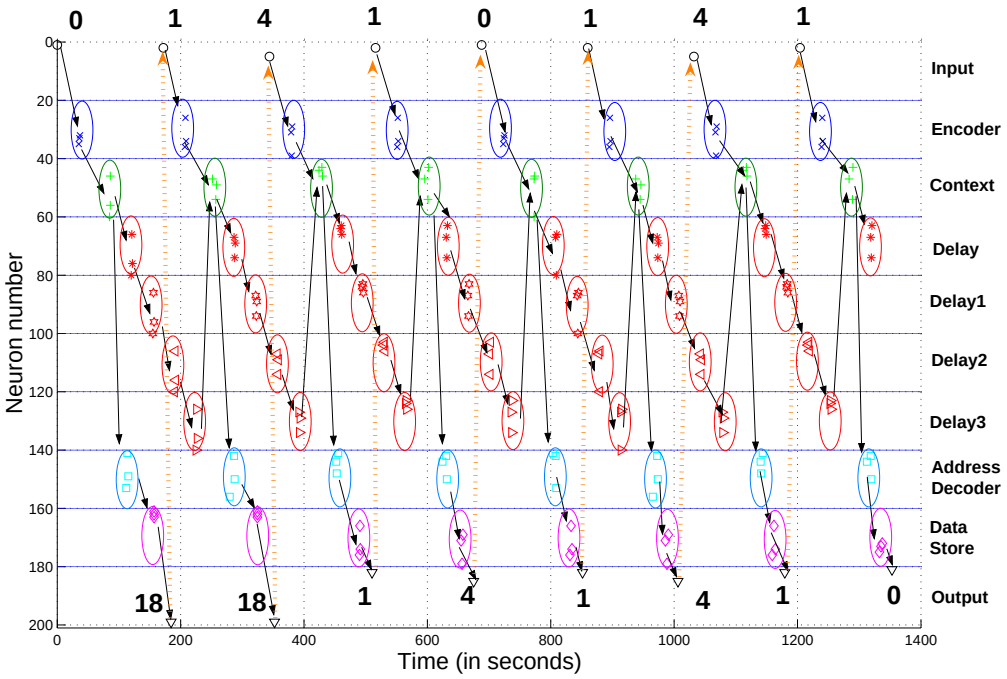

Figure 7.8: Spiking sequence machine output, with different neural layers on the Y-axis and simulated time on the X-axis (actual time to run the simulation is approximately 3 minutes), using an context neural layer model of the sequence machine and four delay feedback layers instead of one. The ellipses denote different layers of neurons, black arrows denote causality, orange arrows prediction. The input sequence is 01410141, which the machine learns to predict correctly after a single pass.

Figure 7.8 shows the output of the simulator (with spike outputs of different neural layers across time) following the context neural layer model.

An ordered 3-of-20 code is used for each layer except the input and output, which follow a 1-of-20 code. The input sequence given to the machine is 0141 0141, the repetition of the 0141 sequence is to enable it to learn it the first time and predict correctly the second time. Here too we can verify that a sequence of 0141 (for which the successor symbol of 1 depends on its context) gets predicted perfectly the second time after it is learnt the first time. For this experiment the

value of λ (context sensitivity factor) is kept at 0.2, and a significance ratio of 0.97 is used throughout.

7.7 Verification of equivalence with the time abstracted model

In order to verify that the output of the spiking neural simulator is exactly the same as the equivalent machine using time abstracted significance vectors, we have developed a Matlab program that takes the same weight matrices as used in the simulator and the same parameters (neural layers, connection structure, values of the significance ratio α and context sensitivity factor λ, N-of-M code, etc) and outputs the order and choices of neural firings (as per the ordered N-of-M code) for each neural layer in the network, and for each wave. The program uses the time abstracted significance vector model to calculate its outputs, instead of using the wheel model as in the spiking neuron simulator.

Comparing the spike timings output by the neural simulator and the Matlab code, we have verified that the time abstracted model performs exactly as the spiking neuron simulator, i.e. the choices and orders of neurons firing in each layer in the system are exactly the same for both the cases, for the experiment repeated with different parameters and even for different models of context encoding including the context neural layer and combined model.

While it is not unexpected to have the simulator and time abstracted model behaving exactly the same, since we have explicitly designed them to do so, it still shows that the time abstracted model can be implemented using spiking neurons, and that the spike bursts in the neural system remain stable and coherent while passing through many layers and also many waves.

7.8 Conclusion

In this chapter, we have described the design and operation of the sequence machine using spiking neurons, and demonstrated that the spiking neural implementation performs equivalently to an implementation using significance vectors. In the next chapter, we present the results of some experiments performed to determine the behaviour of the sequence memory in different conditions.

Chapter 8

Tests on the sequence machine

In the previous chapters, we developed a sequence machine in which the context was stored using a combination of a neural layer and a shift register, and then showed how to implement the machine using the wheel or firefly model of spiking neuron. In this chapter, we first test the tolerance of the spiking neural sequence machine to noise in the spike timings. Then we use the time abstracted model (because it is simpler and easier to use than the spiking neural simulator) to conduct some experiments on the performance of the sequence machine. Finally, we present the results of performance tests done on the machine.

8.1 Performance of the system with non-uniform input spike timings

The experiments we performed using the spiking neural simulator in the previous chapter had the input spikes equally spaced in time. In order to study the behaviour of the system when the spike timings are not uniform, we varied the timings in the following way: we generated random numbers between +1 and -1 from a uniform distribution, multiplied them by a fraction η of the original timing between inputs (say T), and added this value to the input times in the original uniform timing model to create new timings. We varied the spike timings with the sequence 715171517151, whose uniform-timing version was illustrated in Figure 7.7. We repeated the experiment for η =0.001, 0.01, 0.1 and 0.2.

Figure 8.1 shows the output of the experiment when 10% irregularity (η=0.1) is added to the input temporal spacing. We see that the spike bursts from different

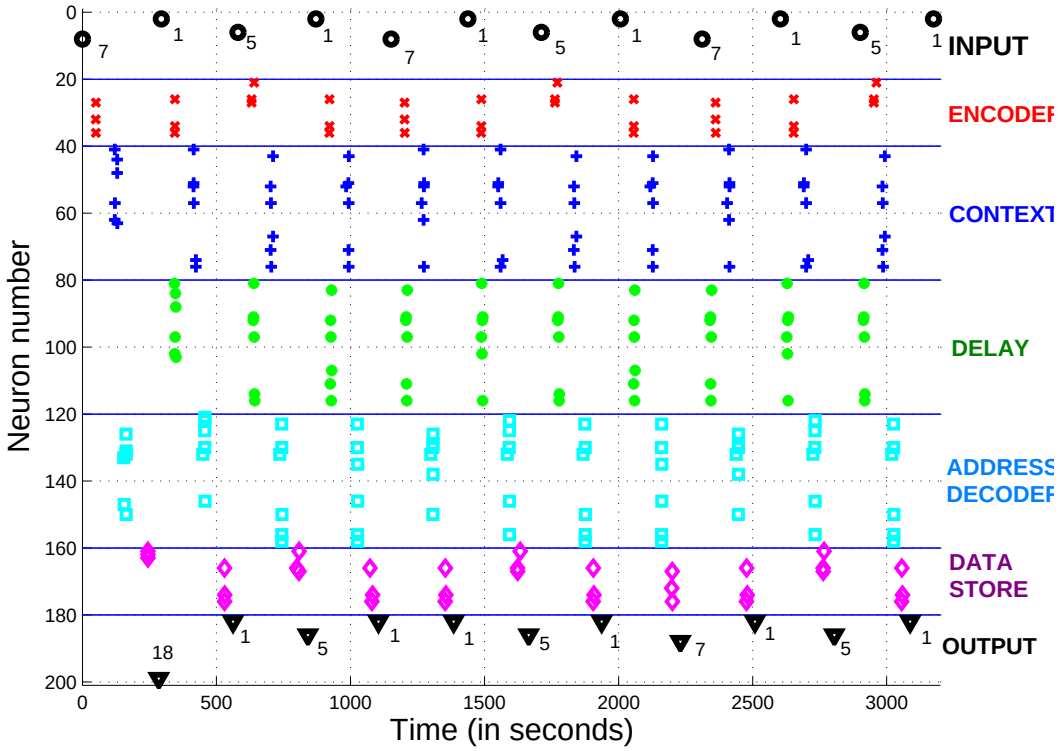

Figure 8.1: Plot of spikes in different layers of the sequence machine for the input sequence 715171517151 against simulated time, when 10% irregularity is added to the input temporal spacing. After the first presentation of 7151 when it learns the sequence, it is able to predict the next symbols in the sequence correctly. The system can cope with the irregularity in spike timings.

layers are still coherent and well-behaved, and the sequence machine predicts the next outputs correctly. Thus the addition of the irregularity in the input spike timings does not have any significant effect on the performance of the machine.

Figure 8.2 shows the output with η=0.2, i.e. 20% irregularity in input temporal spacing. If the input spikes arrive earlier than expected, they will induce encoder spikes to fire, which will then cause firings in the context layer. Normally the context layer output spikes are expected to fire only when both the encoder and delay spikes have arrived, but here the encoder spikes arrive earlier causing the context neurons to fire before the delay spikes, and when the delay neurons fire they cause another set of context spikes. Therefore the spike bursts start to diverge because of the added irregularity, as we can see in the figure after the 715171 input. However, the machine seems to recover in this case with the next wave of inputs, and manages to predict the following inputs correctly. However,

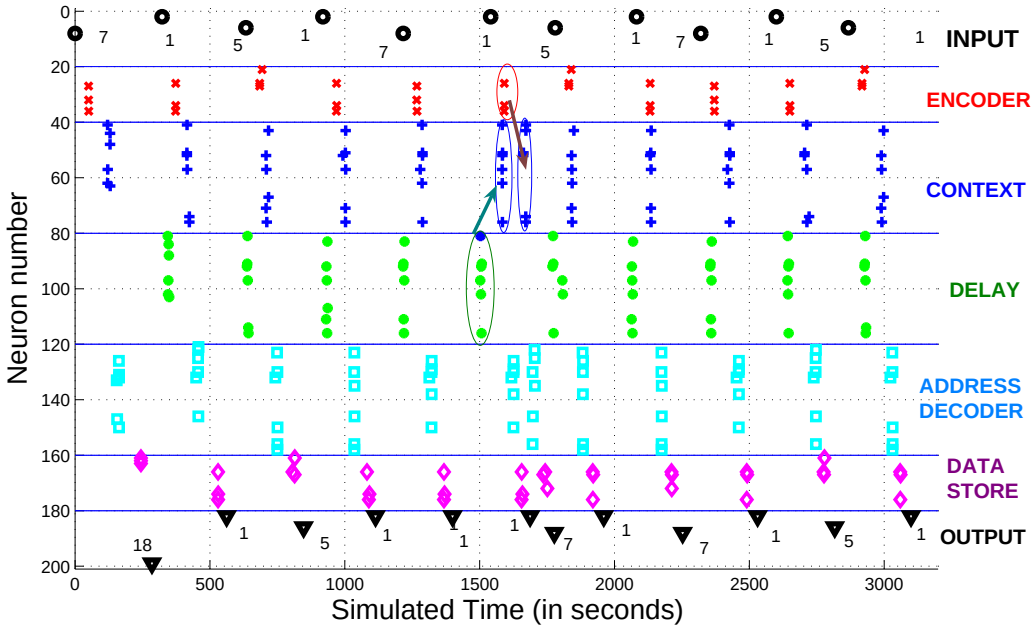

Figure 8.2: Plot of spikes for the input sequence 715171517151 against simulated time, when 20% irregularity is added to the input temporal spacing. Here, the addition of the irregularity disturbs the well-behaved timing behaviour of different layers, causing two bursts to be emitted from the context layer (encircled by the two blue ellipses) in response to the two sets of input spikes from delay (green ellipse) and encoder (red ellipse). However the system recovers and is still able to predict the next inputs 7151 correctly.

with 30% irregularity as shown in figure 8.3, the context layer mixes the spike bursts from encoder and delayed previous context and so machine seems to fail completely. Therefore, we can conclude that the machine is tolerant to a certain degree of irregularity in the input spike timings, and fails only if the irregularity is more than this threshold.

8.2 The time-abstracted and spiking neural implementations

In the time-abstracted model, all inputs and outputs are encoded as ordered N-of-M significance vectors. The purpose of using the vectors was to abstract the firing times of the neurons of a layer during a burst. Using the time-abstracted model simplifies many of the complications that we face when using a model of

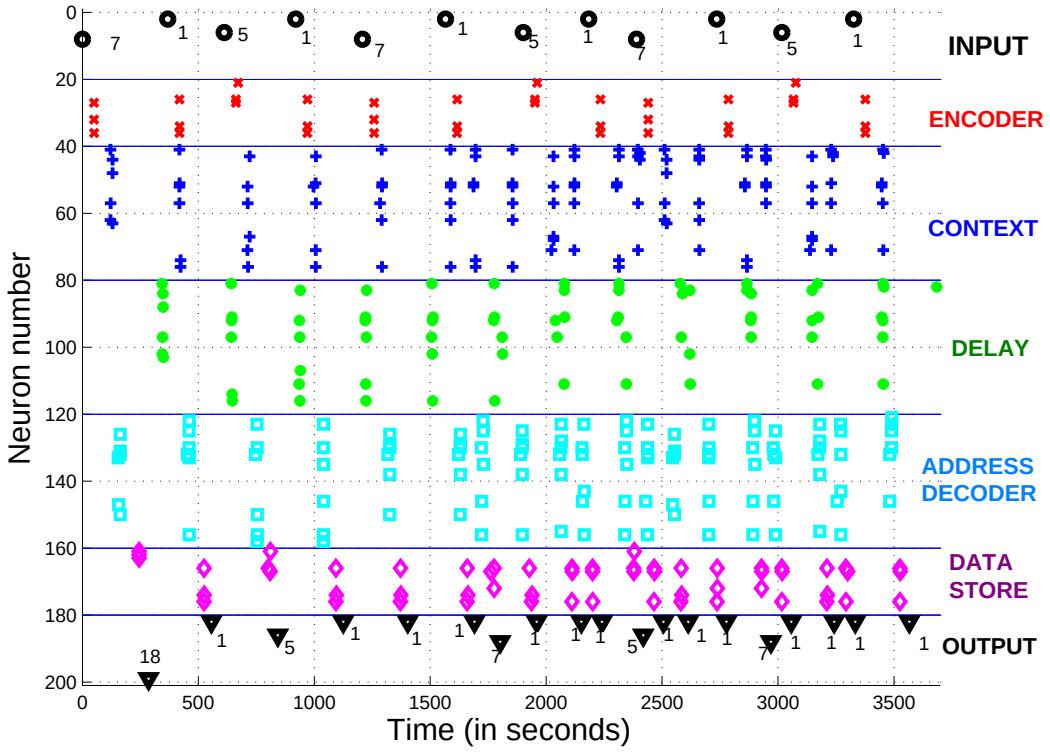

Figure 8.3: Plot of spikes for the input sequence 715171517151 against simulated time, when 30% irregularity is added to the input temporal spacing. Here the context gets confused due to the non-uniform timings of the spike bursts from the encoder and delay layers, and the system does not recover.

spiking neurons. We can input the vectors in functions and perform operations on them without worrying about how and when the spikes are generated, how they get signals, when to start and stop firing and how we ensure two spike bursts do not interfere. We have already shown that simulating the sequence machine using the wheel model of spiking neurons gives equivalent results to the system simulated using the abstracted model. Therefore, from this point on we will use only the time abstracted model for our experiments on the performance of the sequence machine.

8.3 Summary of the model and testing method

In chapter 4 we conducted some tests on the performance of a rank-ordered N-of-M Sparse Distributed Memory (SDM) and showed that it had good efficiency and scalability properties. In the sequence machine, we use the same ordered N-of-M

SDM memory as the associative memory component. We explored in chapter 5 three ways to encode the past history of the sequence in the context using our framework, namely the context neural layer, shift register and the combined model. The last of these combines both the other methods and it is possible to switch from one to the other using the context sensitivity factor λ as the control variable.

During the tests on memory performance, we shall use the framework mentioned in chapter 2 to feed the sequence to the machine, which includes first associating the old context and the input and writing the association to the SDM, forming the new context from the old context and the encoded input and finally generating the output prediction as a function of the new context and the input. This machine can learn a sequence in a single presentation, and when the same sequence is presented again it can predict it correctly. Therefore, to measure the performance of the memory we input each sequence twice into the machine (using the three steps mentioned each time), the first time to write the sequence to the memory and the second time to read it and check if the predicted output is the same as the sequence which was written in earlier. We repeat this for different lengths of the sequence (clearing the memory after each sequence of a specified length) starting with a sequence of length 2, and plot the graph of the number of symbols in the sequence predicted correctly verses the length of the sequence. We perform this experiment for the three kinds of model we described (the context neural layer, the shift register and the combined model) and also varying different memory parameters to observe how the memory behaves in different circumstances.

In the sequence memory experiments, we perform the following steps:

1. We **initialise the sequence memory** by generating the connection weight matrices for various components in the system. We generate the weight matrices of the address decoder layer, the context layer in the neural layer model, and the context scrambler (the connection weights between the delay layer presenting the old context to the new context layer) and input expander (the connection weights from the input to the new context layer) in the combined model.

2. We then **generate a random sequence** of specified length (say l) composed of symbols from the alphabet. The symbols forming the sequence

are chosen randomly from an uniform distribution (meaning that each of the symbols in the alphabet has equal possibility of being chosen) in most experiments. In some experiments we vary the distribution and note the performance of the memory with different distributions.

3. **Training phase**: We feed this generated sequence to the sequence machine, symbol by symbol, where the associations between each encoded input symbol and the present context are written to the memory as per the algorithm described. The machine generates an output prediction for each input symbol, but since this is the training epoch these predictions are ignored.

4. **Testing phase**: We now re-input the sequence generated in Step 2 to the machine, one symbol at a time. This time we expect the predictions for the next symbols to be correct. Therefore, with each input symbol, we compare the output (which is a prediction for the next input symbol) to the next symbol in the sequence. When the complete sequence is finished, we count the total number of symbols in the sequence that are predicted correctly.

5. We **vary different parameters and repeat steps 1-3**, each time counting the number of symbols predicted correctly, and clearing the sequence memory between trials. We then plot the performance of the machine.

8.4 Testing method in related approaches

The testing method in most other approaches to sequence learning (such as the approaches of Hochreiter [34], Berthouze [12], Tino [30], Elman [23] etc) followed a common procedure: they first trained the model over sequences generated by a grammar or time series for a number of epochs or training cycles. Then they used the same or a different test data set generated by the same grammar and tested the trained network to correctly predict or classify the test data on the basis of what it learnt during training.

For example, in Schmidhuber's approach of training a machine to learn an embedded Reber grammar ([34]), variable length strings were generated using this grammar. 256 such random strings from the grammar were generated for the training set and 256 different strings for the testing set. Then the performance of the system was measured and compared with other common approaches such as

the Elman net trained by the Elman procedure [23]. In Tino's predictive model built using a method similar to a variable length markov model ([78]), the model was tested by using the daily values from the Dow Jones Industrial average over a period of time, the data being transformed into a time series. The model was trained using a number of training cycles and tested on other data. In Berthouze et al's approach [12] to context dependent sequence learning, the testing was done on sequences of specific length generated randomly by a grammar used in a game called Tekken. The grammar was like a tree containing a character on each node, and a sequence could be generated by traversing the tree. Sequences having specific properties (such as certain characters being repeated) were first generated using the grammar and the system was then trained and tested on these sequences to see if it could remember the grammar or recall any sequence generated by this grammar.

In most of the above tests, the objective was to make the machine learn the generating process or grammar during training, so that it could predict correctly the next elements in the sequence generated by the same process. However, as mentioned before, we make no assumptions about the generating grammar and our sequence machine is expected to learn and predict any sequence of symbols that is explicitly presented to it. Therefore, in our tests we generate random sequences from a finite alphabet and feed them twice to the machine, the first time for learning the sequence and the second time for testing. During the testing, we evaluate how many predictions are correct by comparing them to the original sequence.

8.5 Setting the parameters

The sequence machine has a large number of parameters which can be varied. To facilitate a fair comparison of the three models (the context neural layer, the 2-shift register and the combined model) we shall keep most of the parameters the same across the three models and only vary those that we are actually measuring. The parameters we keep constant over the three models of context encoding are the SDM memory parameters (the dimensions of N-of-M, the number of address decoders, significance ratio, etc) in order to study the performance for a specific memory. The parameters we vary are the context sensitivity factor and the size of the sequence. To be fair in comparing the three different kinds of models, we

optimise each of them separately (with respect to the variable parameters) before comparing them.

The parameters that we keep fixed for the sequence machine performance tests are as follows:

- The significance factor α is kept as 0.99, which is the optimal value used in the SDM memory in Chapter 4
- We use an SDM with dimensions 22-of-512 as the d-of-D (instead of using 11-of-256 as in Chapter 4) and 16-of-4096 as the w-of-W code for the address decoder.

  The reason we used a 22-of-512 code instead of 11-of-256 for the address and data inputs to the SDM is that in the sequence machine, the context vector is input as the address to the address decoder layer of the SDM. In the 2-shift register, the context output (22-of-512) has to be double of the input vector (11-of-256) because the look-back time window size is 2. To compare the 2-shift register with the neural layer and combined models, we kept the size of the context vector same (22-of-512) in all the three models.

The parameters we vary during the memory performance tests are as follows:

- Size A of the alphabet from which the input sequences are created
- The distribution of the sequences from the alphabet. For example we may have binary sequences with uniform distribution of 1's and 0's, or we may have non-uniform distributions such as 0.9 probability of 1's and 0.1 probability of 0's. We have used a parameter p to denote the distribution of the first symbol in the alphabet (assuming the remaining symbols of the alphabet are chosen randomly from an uniform distribution). For example, if we have a binary alphabet (0's and 1's) with distribution p, it would mean that 0 is selected with a probability of p and 1 with a probability of (1-p)
- Length of the input sequences created using a given alphabet and a given distribution
- The context sensitivity parameter λ, denoting the relative influence of the past context and the input in determining the new context

- The method used to generate the new context from the past context and input. As already mentioned, we have used three models, the neural layer, shift register and combined models

8.6 Tests of the memory performance

We conducted experiments on the sequence machine, to analyse its performance against different criteria. The tests are to establish what kinds of sequences the memory can store and how well it performs the storing and recalling sequences of different lengths. Here we are especially interested in sequences which have characters in common or sequences with repeated characters. The effect of the context sensitivity on the types of sequences the machine can recognise is also an interesting observation. In these tests, we compare the memory performance of the predictive on-line sequence machine using three ways to encode the context, namely the shift register, context neural layer and combined model of the neural layer and shift register.

8.6.1 Effect of the context sensitivity parameter λ in the neural layer and combined models

In this experiment, we vary the context sensitivity parameter λ for the different models of context encoding, in order to determine the optimal value of λ. We keep the sequence length fixed at 2000, and alphabet size fixed at 10.

The results are shown in figure 8.4. As we can see, the optimal value of λ for the context neural layer model is 0.2 and for the combined model is 0.9, for a sequence length of 2000 drawn from an alphabet size of 10. Hence, we shall use these optimal values in future experiments where needed.

We conducted another experiment to see how the memory performance in the combined model varied with λ when the input sequence length was also varied between 100 and 500, with the alphabet size kept fixed at 10.

The results are plotted in figure 8.5. We see that the number of symbols in the sequence that are recalled correctly rises with an increase in the input sequence length. For an input sequence length of 500, using λ values of 0.7, 0.8 and 0.9 give nearly the same number of correct recalls, which is the best among all λ values tested. The memory performance worsens appreciably for λ=1.0 and higher, as

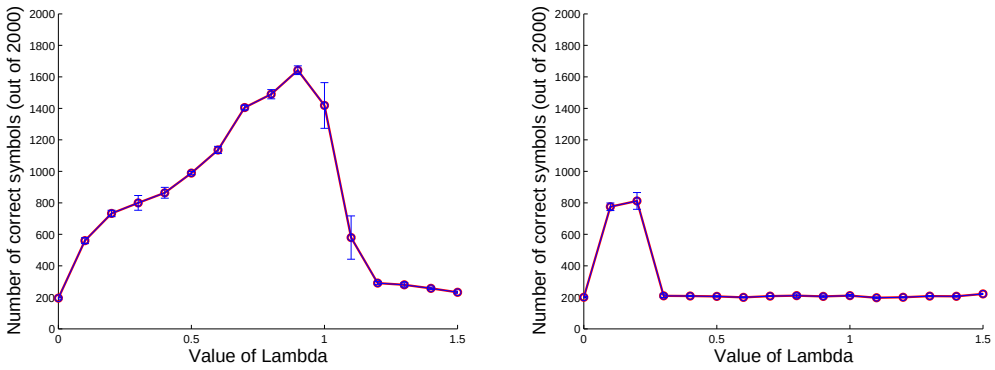

Figure 8.4: (a) Plot of the sequence machine performance with the context sensitivity λ varying from 0 to 1.5 for the combined model, averaged over 5 trials. The input sequence length is kept fixed at 2000 and alphabet size at 10. A value of λ=0.9 gives the best performance. (b) Performance of the sequence machine with varying λ for the context neural layer model, with input sequence length also fixed at 2000. Here the performance is optimal for λ=0.2

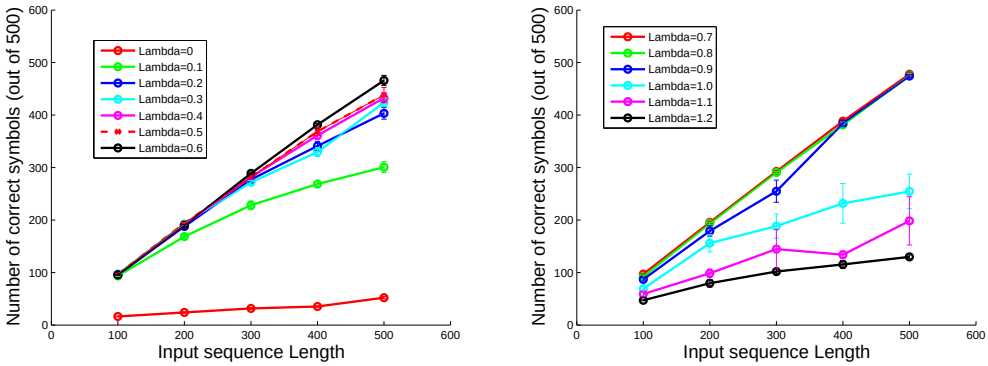

Figure 8.5: (a) Plot of the sequence machine performance with the context sensitivity parameter λ varying from 0 to 0.6 for the combined model, averaged over 5 trials. The input sequence length is varied from 100 to 500, and the alphabet size is 10. (b) Performance of the sequence machine with varying input sequence lengths and λ varied from 0.7 to 1.2 for the combined model.

the past context becomes more important than the present input in determining the new context. In the earlier experiment plotted in figure 8.4, using λ=0.9 gave the best performance for a sequence length of 2000, which is more than 500 and so is more reliable. Hence we shall use the value of 0.9 as the optimal λ value for the combined model in future experiments.

We then tested how the memory performance changes with varying lamba and input sequence length for a binary alphabet. We varied the context sensitivity

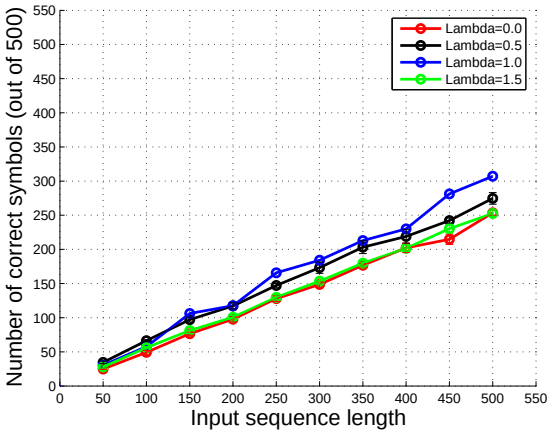

Figure 8.6: Performance of the sequence machine with symbols drawn with a uniform distribution from a binary alphabet, averaged over 5 trials and plotted with symmetric error bars. The input sequence length is varied from 50 to 500.

factor λ from 0.0 to 1.5, and the input sequence length from 50 to 500. The results are plotted in figure 8.6. The chance level for a sequence of length L drawn from a binary alphabet with uniform distribution (1 and 0 chosen with equal probability) is L/2. As we can see from the figure, the performance of the machine at λ=0.0 (when the new context is determined based solely on the input) and λ=1.5 (when the past context is more important than the present input in determining the new context) is almost equal to the chance level. The performance is slightly better at λ=0.5 and is best at λ=1.0. The performance is generally quite poor, which is expected since there are bound to be many repeated subsequences and ambiguities when the alphabet size is 2 and the sequence length is significantly larger than 2.

8.6.2 Effect of alphabet size on memory performance of the combined model

In this experiment, we vary the alphabet size for the combined model, keeping the context sensitivity parameter fixed λ at 0.9 (the optimised value) and see how it effects the memory performance.

Symbols constituting the sequence are selected from an uniform distribution. The sequence length is kept at 500. The alphabet size is varied from 2 (representing a binary alphabet) to 15. For symbols chosen from a binary alphabet, the chance level score for retrieval (if the symbols were chosen randomly) would

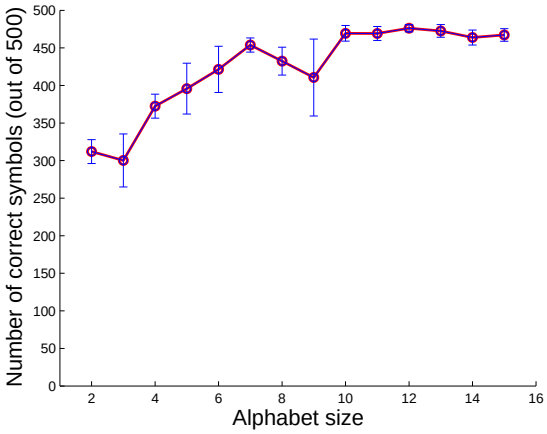

Figure 8.7: Performance of the sequence machine with varying alphabet size A (from which symbols are chosen with a uniform distribution) ranging from 2 to 15, averaged over 5 trials and plotted with symmetric error bars. The input sequence length is 500. The machine uses ordered N-of-M SDM parameters d-of-D 11-256 and w-of-W 16-4096, with context sensitivity factor λ kept at 0.9 and significance ratio α kept at 0.99

be half of the sequence length, i.e. 250 out of 500.

The results are shown in figure 8.7. We see that for a low alphabet sizes of 2 and 3, the number of symbols out of 500 (which is the sequence length) correctly recalled is low, the reason being that the number of repeated symbols in the sequence is far too many in case of sequences generated from small alphabet sizes. However, performance gradually improves as the alphabet size increases, reaching saturation close to perfect recall for alphabet size 10 or higher. The reason for this could be that as the alphabet size gets bigger, the probability of symbols getting repeated in the generated sequence decreases.

8.6.3 Effect of λ and distribution on memory performance

In this experiment, we vary the distribution of the symbols from the given alphabet constituting the sequence, and see how it effects the memory performance, as a function of the context sensitivity factor λ.

In the experiments so far we have only dealt with uniform distributions, where each symbol in the alphabet has equal probability of being selected to form the sequence. If A is the alphabet size, a symbol s in the alphabet can be denoted by the following relation:

$$s \in \{s_1, ..., s_A\} \tag{8.1}$$

Let $p(s_n)$ be the probability that symbol s_n is chosen from the alphabet. For an uniform distribution, the probability of every symbol getting chosen is equal. Therefore

$$p(s_n) = 1/A \tag{8.2}$$

In a skewed distribution, let us define the parameter pdist as the probability of the first symbol in the alphabet getting selected, assuming the rest of the symbols are chosen with equal probability. Therefore, $p(s_1) = pdist$. Let q denote the probability of each of the remaining symbols getting selected. Hence, $p(s_2) = p(s_3) = .. = p(s_A) = q$

The relation between p and q in such a skewed distribution can be expressed as follows:

$$pdist + (A-1)q = 1 \tag{8.3}$$

since the total probability of any symbol getting selected is 1.

Solving the above equation, we get the following expression for q in terms of p:

$$q = \frac{1 - pdist}{A - 1} \tag{8.4}$$

For example, for a binary alphabet (A=2), $q = 1 - pdist$. For an alphabet of size 10, $q = (1 - pdist)/9$.

We now perform some experiments using skewed distributions, characterised by the parameter pdist.

We vary the context sensitivity factor λ between 0.0 and 1.5, with a step of 0.5. The sequence length is kept constant at 50.

We perform this experiment first with a binary alphabet. There are only two possible symbols in the alphabet, 0 and 1. We first generate sequences with symbols chosen from an uniform distribution, i.e. the probability of symbol 0 getting chosen is the same as of symbol 1 (q=pdist=0.5). We then repeat the experiment with pdist=0.1 (meaning p(0)=0.1 and p(1)=0.9) and pdist=0.3.

The results are plotted in figure 8.8.

We see that the skewed distribution performs slightly better than the binary distribution, for a higher value of λ (close to 1). This is because in a binary sequence with uniform distribution, the probability of symbols 0 and 1 getting

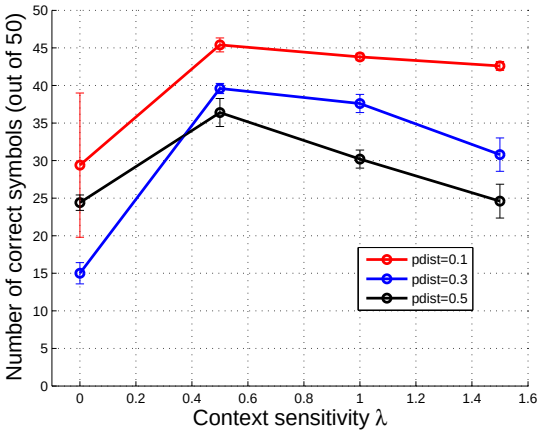

Figure 8.8: Performance of the sequence machine with symbols selected from a binary alphabet with varied distribution factor pdist, averaged over 5 runs with error bars and plotted against the context sensitivity factor λ. The sequences are of length 50. Distribution factor pdist of 0.5 represents uniform distribution and 0.3 and 0.1 are skewed distributions.

selected are equal, so there is less possibility of the sequence being distinguishable to be recalled. With skewed distribution, there are more 1's than 0's, so there will be more repeats but also the sequence will have more chance to be remembered.

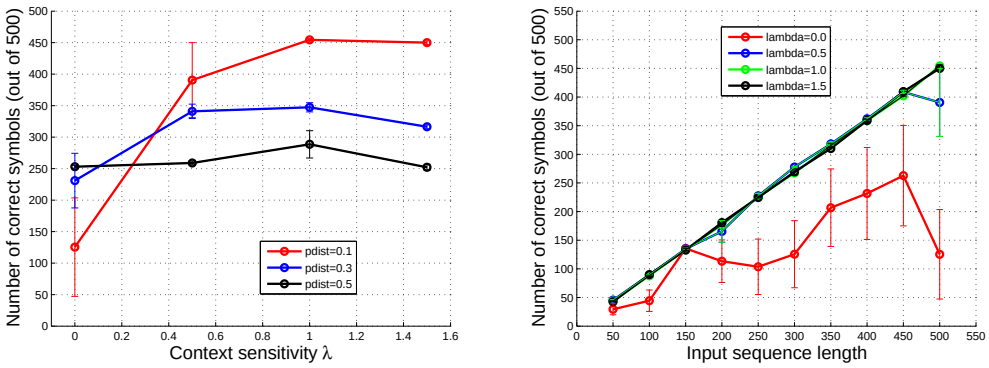

Figure 8.9: (a) Performance of the sequence machine with symbols selected from a binary alphabet with varied distribution factor pdist, averaged over 5 runs with error bars and plotted against the context sensitivity factor λ. The sequences are of length 500. Distribution factor pdist of 0.5 represents uniform distribution and 0.3 and 0.1 are skewed distributions. (b) Plot of the performance for a pdist of 0.1 for a binary alphabet, with sequence length varied from 100 to 500.

Figure 8.9(a) plots the performance of the sequence machine with symbols chosen from a binary alphabet for a sequence length of 500, with lambda varied

from 0.0 to 1.5 and pdist varied between 0.1 and 0.5. We see that a skewed distribution with pdist=0.1 gives optimal performance, which is much better than an uniform distribution with pdist=0.5. Figure 8.9(b) varies the sequence length for the same, keeping pdist at 0.1. We see that apart from λ=0.0, which performs quite poorly, the rest of the λ values perform very well.

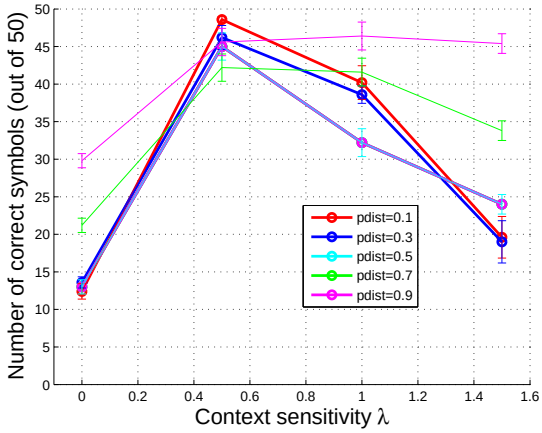

Figure 8.10: Performance of the sequence machine with varied distribution factor pdist, averaged over 5 runs and plotted against the context sensitivity factor λ, varied between 0.0 and 1.5. The generated sequences are of length 50, with symbols selected from an alphabet of size 10. The value of pdist=0.1 represents uniform distribution and the values of 0.3, 0.5, 0.7 and 0.9 are skewed distributions.

We then repeat this experiment for an alphabet of size 10, and the distribution parameter pdist given values 0.1 (representing the uniform distribution where the first symbol, as well as the remaining symbols, are chosen with 0.1 probability), 0.3, 0.5, 0.7 and 0.9.

The probability of choosing a specific symbol out of the alphabet of 10 symbols can be calculated from the equation above. For example pdist=0.5 represents a skewed distribution where symbol 0 is chosen with probability 0.5 and the remaining symbols 1 to 9 are each chosen with a probability of 0.5/9=0.056.

The results are plotted in figure 8.10.

We see that the most skewed distribution (most dominated by the first symbol) at pdist=0.9 performs generally the best across different values of λ, except at the case of λ=0.5, when the uniform distribution (pdist=0.1) performed the best.

We repeated the same experiment (varying both pdist and λ for an alphabet

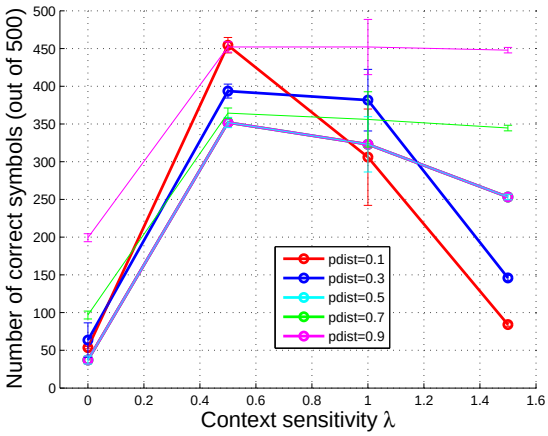

Figure 8.11: Performance of the sequence machine with varied distribution factor pdist, averaged over 5 runs and plotted against the context sensitivity factor λ, varied between 0.0 and 1.5. The generated sequences are of length 500, with symbols selected from an alphabet of size 10. The value of pdist=0.1 represents uniform distribution and the values of 0.3, 0.5, 0.7 and 0.9 are skewed distributions. Using pdist=0.9 gives the best performance.

size of 10), this time with the input sequence length kept constant at 500. The results are plotted in figure 8.11. We see that here again, using a value of pdist=0.9 (representing the most skewed distribution) gives the best results across different values of λ.

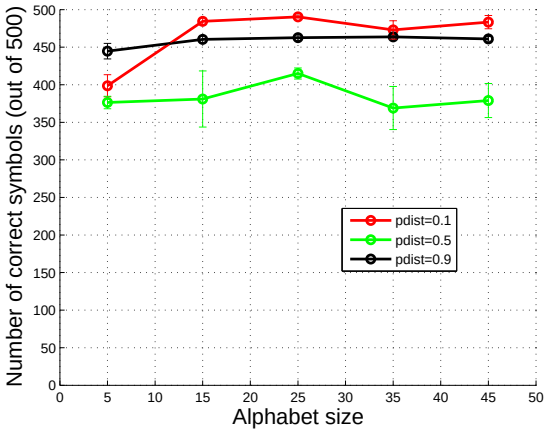

Figure 8.12: Plot of the sequence machine performance with varying distribution parameter pdist for different alphabet length A, averaged over 3 runs. Context sensitivity parameter λ is kept at 0.9. Input sequence length is 500.

We then conducted another experiment to measure how the alphabet size

effected the memory performance at different values of the skewed parameter pdist. The results are plotted in figure 8.12. The context sensitivity factor λ is kept constant at 0.9, although the optimal value of λ as 0.9 was tested only for an alphabet length of 10 and for a uniform distribution. We found that pdist of 0.5 performed the worst, regardless of alphabet size. The machine using pdist = 0.1 performed worse than the one using pdist=0.9 for alphabet length A=5, but performed better at higher alphabet lengths. A reason for this could be that at alphabet length A=5, pdist (the probability that the first symbol gets chosen) for an uniform distribution would be 1/5=0.2, so using pdist=0.1 for alphabet length 5 means less than average probability of the first symbol getting selected. We saw earlier that more skewed distributions (higher pdist than average) generally performed better than uniform distributions.

8.6.4 Comparison of different models of context encoding

In this experiment, we compare the three models of context encoding in the sequence machine (i.e. neural layer, shift register and combined model) and analyse their performance for different sequence lengths from 100 to 2000. The alphabet size is 10, and symbols are chosen from a uniform distribution. In this experiment, we use the optimised values of context sensitivity parameter λ for the context neural layer (optimised value=0.2) and combined model (optimised value=0.9). The shift register does not use the λ parameter.

Figure 8.13 shows the results of the experiment. We see that the combined model performs the best of the three consistently, even for sequences of the length of 2000, which is 20 times the alphabet size of 10 (causing many symbols to repeat in the sequence).

The 2-shift register performs worst of the three, possibly because its maximum look-back is 2, and with higher sequence lengths, there is an increasing possibility of more than 2 characters been in common within the generated sequence, leading to ambiguities in the recall.

8.6.5 Comparison of the three models while varying the alphabet size

In this experiment, we compare the performance of the neural layer, shift register and combined model for different alphabet sizes. The input sequence length is

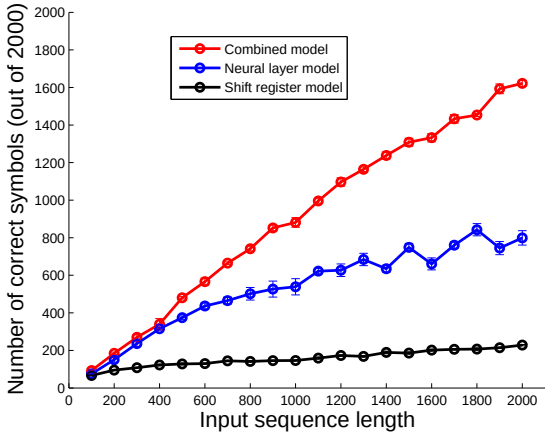

Figure 8.13: Performance of the sequence machine with three kinds of context encoding : context neural layer, 2- shift register and the combined model. The alphabet length is 10, and the sequence length is varied between 100 and 2000, with steps of 100. The optimised combined model (with λ=0.9) performs better (least number of errors) than the others.

varied from 100 to 500, and symbols constituting the sequence are chosen from the alphabet with a uniform distribution.

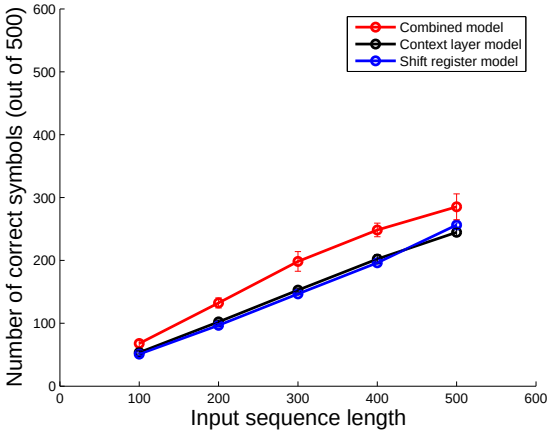

Figure 8.14: Plot of the sequence machine comparing the performance of the shift register, neural layer and combined models of context encoding, when symbols are chosen from a binary alphabet with an uniform distribution

Figure 8.14 compares the three models when using an alphabet length of 2, averaged over 5 runs. The input sequence length is varied from 100 to 500. The chance level (the expected value of the number of output symbols correctly recovered if the output symbols are generated randomly from the alphabet) is half

of the input sequence length for a binary alphabet. We see that the combined model is better than the context neural layer model, whose performance is nearly the same as the shift register model. However, all three models perform quite poorly, because there is a high probability of ambiguities in determining the next output symbol due to many repeated symbols in a sequence generated from a small alphabet.

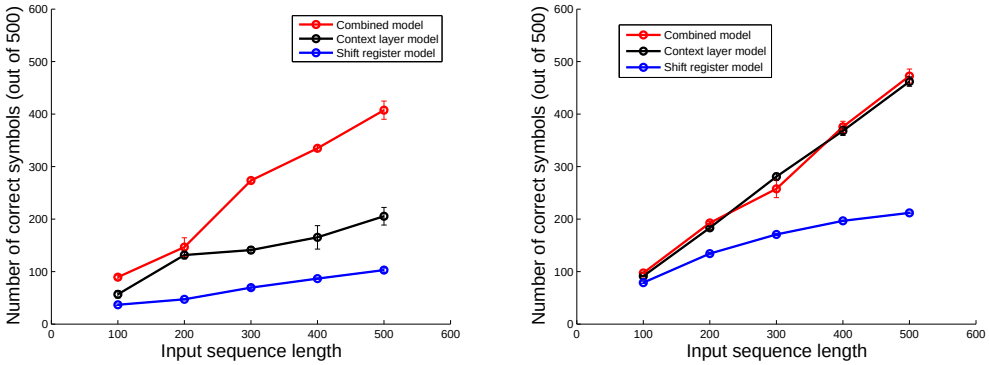

Figure 8.15: (a) Plot of the sequence machine performance for the neural layer, shift register and combined models, with input sequence length varying from 100 to 500 and alphabet size of 5, averaged over 5 runs. (b) Performance of the sequence machine for the three models when the alphabet size is 15. Performance of the combined and context layer models is close to perfect recall.

Figure 8.15 compares the three models when using varying alphabet lengths of 5 and 15. We see that for an alphabet of 5, the combined model performs much better than a context layer model (diverging more for a higher sequence length), which in turn performs better than a shift register. For an alphabet size of 15, the combined and neural layer models perform nearly the same, which is close to perfect recall and much better than that of the shift register. The reason is that the memory performance saturates at larger alphabet sizes, since there are fewer ambiguities caused by repeated symbols.

8.6.6 Investigation of the effect of convex combination Λ on memory performance

In this experiment, we plot the performance of the sequence machine for the combined model using the convex combination Λ as the context scaling factor. We had mentioned the use of Λ in chapter 5. The input sequence length is varied

between 50 and 500. Symbols are chosen from an uniform distribution from an alphabet of size 10.

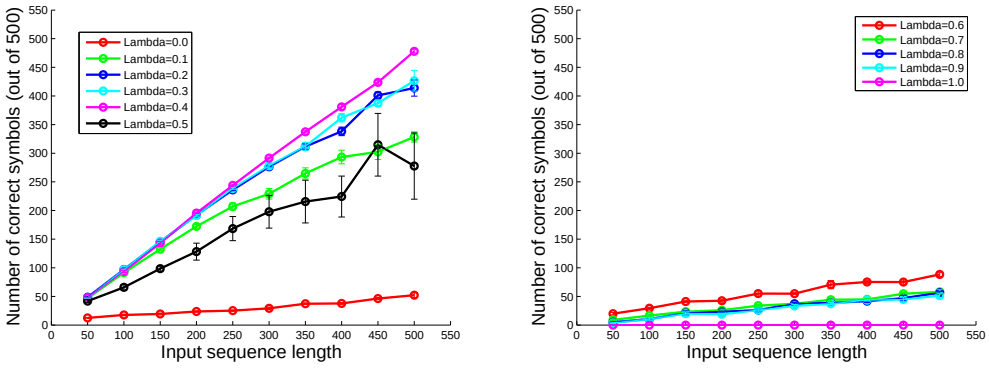

Figure 8.16: (a) Plot of the sequence machine performance for the combined model for Λ varying from 0.0 to 0.5, with the input sequence length varying from 50 to 500 and alphabet size kept constant at 10, averaged over 5 runs and plotted with error bars. (b) Performance of the sequence machine with Λ varying from 0.6 to 1.0.

The results are shown in figure 8.16. As we can see from the figure, the memory performance is quite poor at Λ=0.0 (corresponding to λ=0.0), peaks at Λ=0.4 (corresponding to λ=0.67), and becomes worse for Λ=0.5 (corresponding to λ=1.0) and higher, going down to nearly 0 (hardly any symbol correctly recovered) at Λ=1.0 (where the input plays no part in determining the new context). This is roughly equivalant to the performance we plotted in figure 8.7 (where we plotted the performance varying both λ and input sequence length) for the corresponding values of λ and Λ.

8.7 Conclusion

From the tests on the sequence machine described in this chapter, we can conclude the following:

Tolerance of the spiking model to irregularity in input temporal spacing: the spiking model is tolerant to an irregularity of 20% in input spacing, beyond which the context is unable to distinguish the next outputs correctly.

Optimal value of context sensitivity parameter λ: for the combined model, the optimal value for a uniform distribution is 0.9, when the input sequence length was varied. For the context neural layer the optimal λ value is 0.2.

Optimal value of λ for varying distribution: using a distribution parameter pdist of 0.1 gave best results for a binary sequence, and a pdist value of 0.9 gave the best results for sequences chosen from an alphabet size of 10. In both cases, using the most skewed values of pdist gave the best results.

Comparison of the context neural layer, shift register and combined model: the combined model performs better than the neural layer and shift register consistently across different alphabet sizes, when the sequence length was varied. The context layer model performed better than the 2-shift register.

Optimal value of the convex combination parameter Λ across different input sequence lengths is 0.4. Using values of Λ from 0.6 to 1.0 perform quite poorly.

In the next chapter, we shall conclude the dissertation by discussing what we have achieved and the ways in which this work can be improved.

Chapter 9

Conclusion

9.1 Summarising the research

In the previous chapters the design and implementation of a sequence machine built out of spiking neurons was described. The implementation was done in the following steps: First of all, we proposed a framework for on-line predictive sequence learning with asynchronous updating of the sequence context in cases where the generative model of the sequence was unknown (in chapter 5), and used it to build a sequence machine that employed a rank-ordered N-of-M SDM as the associative memory, along with a combination of an context neural layer and shift register to encode the context or past history of the sequence (in chapter 5). We did some tests to measure the performance of the ordered N-of-M SDM memory according to various parameters (in chapter 4). We then developed a sequence machine that used an abstraction of timing information of a spike burst by using significance vectors, and later developed an equivalent model implemented using spiking neurons following the *wheel model* (also known as the *firefly model*) in chapter 6. We described issues related to the stability and coherence of spike bursts emitted by various neural layers in the system, and examined in detail the timing constraints in the system. Then we simulated a system implemented in spiking neurons that followed these timing constraints and showed that it could successfully learn a sequence online. Finally, we performed some tests on the sequence machine to measure its performance qualitatively as well as quantitatively in learning sequences.

Throughout this dissertation, we have examined a variety of issues relating to spatial encoding of temporal information, distance between two neural codes,

efficiency of associative memories, dynamics of spike trains and building a high-level model out of asynchronous spiking neural components.

9.2 Achievements and contributions

In chapter 1 we had laid out some research questions to answer during the course of the dissertation. In this section we shall examine to what extent have we answered those research questions.

- We have shown that it is feasible to build a sequence machine out of spiking neural components, that can learn and predict sequences of symbols online, each symbol being encoded as a burst of spikes emitted by a neural layer. We have shown that a system can be built that can stably and reliably transmit a burst of spikes across different layers in the system. We have also shown that having unevenly spaced input spikes does not significantly impede the performance of the system to the extent of 20% noise, beyond which the context gets confused. (in Chapter 8)

- We have shown how a system implemented in spiking neurons (using a simple linear spiking neural model) can perform exactly as an equivalent system implemented using the temporal abstraction (as a vector of significances) from a burst of spikes encoded using rank-order codes. (in Chapter 7)

- We have proposed a way to build a system that can combine the features of the neural layer and shift register models, by using a control variable λ to modulate the relative importance of the context, (in Chapter 5)

- We have studied and proposed first solutions to different issues relating to modelling with spiking neurons. In particular, we have shown through simulation that it is feasible to have a stable and coherent burst of spikes propagating through many neural layers, as long as the noise level is less than 20%. (in Chapter 7)

- We have proposed an asynchronous update rule for updating the various components of the system during online predictive sequence learning (in chapter 5)

- We have characterised the sequence machine performance for various situations and found the optimal value of context sensitivity parameter lambda for different types of uniform distributions (with parameter p), different lengths of the alphabet and different sequence lengths. (in chapter 8) We found that the combined model performs better than the context neural layer and shift register models for a range of input sequence lengths and alphabet sizes, that the optimal value of the context sensitivity parameter λ was 0.9 for the combined model and 0.4 for the neural layer model across different input sequence lengths, the optimal context sensitivity convex combination Λ was 0.4, and that using a more skewed distribution gave a better performance for a binary code (optimal pdist=0.1) and a code with alphabet size 10 (optimal pdist=0.9) than the corresponding uniform distribution. We also found that the sequence machine implemented with spiking neurons was tolerant to 20% irregularity in the temporal spacing of input spikes. (in chapter 8)

Thus we have successfully accomplished most of the aims of the work, which were laid down at the start of the dissertation.

We have engineered an online predictive sequence machine system from the top down with spiking neurons. Thus, we have built a working system that accomplishes a high-level task within all the constraints mentioned. Our model can be thought of as a combination of long-term and short-term memory, as an example of an application that can be implemented using spiking neurons or as an associative memory implemented in spiking neurons using a localised learning rule where the items to be associated are not simultaneous but arrive at different times.

9.2.1 Answering thesis questions

In section 1.4 of chapter 1, we had proposed a number of thesis questions. Here we mention to what extent have we been able to answer those questions during the course of this research.

1. What are the different ways in which the performance of an associative neural memory can be measured? In chapter 4 in section 4.3.1, we mentioned three ways to characterise the performance of an associative memory: its perfect match capacity, average recovery capacity and information density.

We then performed experiments on the ordered N-of-M SDM to determine whether the memory exhibits the properties of scalability, error tolerance and what is the effect of ordering on the memory performance.

2. What is a useful way to represent the concept of 'distance' or similarity between symbols encoded as vectors (which in turn represents a sequence of firing times by a neural layer) in the system? In chapter 4 , in section 4.1.2, we described the concept of distance as measured by the cosine or the normalised dot product, for which we plotted the memory experiments in chapter 4.

3. In a given sequence, how can the entire past history or context of the sequence be encoded in the best possible way in a vector of finite length? In chapter 2, section 2.10, we discussed different models to encode the past context, including plate's holographic method [62], shift register model [32] and elman networks [23]. In chapter 5, section 5.2, we proposed the combined model of context encoding.

4. How can we build a system where the spike firing times in a layer of neurons are represented and stored spatially (i.e. in the connections between neurons) in such a way that we can learn the firing order and reproduce it when needed? In chapter 6, section 6.4, we showed through simulations that it was possible to have a stable burst propagation across many layers of a feed-forward system of neurons by using a combination of feed-forward shunt inhibition and feedback reset inhibition. In chapter 7, we implemented such a system using spiking neurons in section 7.6, and showed in chapter 8 section 8.1 that such a system was resistant to 20% irregularity in input spike timings.

5. What are the factors influencing the stability of a burst of spikes passing through neural layers and what are the principles of designing a system to be more stable? In chapter 6, section 6.4, we discussed different issues that influence burst stability.

6. What kind of spiking neural model should we choose for implementing specific networks, and why? In chapter 6, in section 6.1 we discussed various considerations while choosing a suitable spiking model. In section 6.2, we proposed a novel RDLIF model that was like the standard LIF model [53]

but had richer dynamics and in appendix A, we showed how the RDLIF model was unsuitable to implement our system. After that, in section 6.3 we chose a simpler linear model called wheel model and showed in chapter 7 section 7.6 that the high level sequence machine could be simulated by spiking neurons following the wheel model.

7. How best can we characterise the performance of a sequence memory? In chapter 8 section 8.6 we discussed various ways to characterise the performance of a sequence machine and performed experiments to learn how our model performed under various conditions. We offered guidance on what kind of lambda values would be suitable to use for varied distributions of the sequences of different lengths generated from symbols taken from different sizes of alphabets.

9.3 Analysis of the model

In this section, we analyse the sequence machine model in more detail, by reviewing the strengths and weaknesses of the model.

9.3.1 Strengths of the sequence machine

The sequence machine is a distributed and flexible system that can be implemented using spiking neurons. Accordingly, it has all the advantages that come from using a neural memory and a memory implemented using asynchronous spiking neural components, such as the ability to keep functioning when the input spike timings are not uniform. It has both short-term (the context layer) and long-term (hetero-associative memory) components. The machine should be useful wherever there is a need for an application that has similar features. Some possible applications of the model will be examined in a later section. However, it must be noted here that although the application space is huge, much work is required before the present model can be applied to any of these applications.

The other strengths of the model lie in the development of the framework for on-line sequence learning, which can be adapted to most learning rules. The system can also be adapted to model parts of the brain or different natural or biological phenomena having similar structures to our design using spiking neurons.

We can make application-specific modifications to the sequence machine (while keeping its structure the same) to enhance its capability and functionality. For example, once the machine has been trained on a long sequence of characters such as the entire text of a dictionary, we can fix the weights and turn off any further training, although it will still be able to form a context from the past symbols and predict the next symbols in the sequence. We can thus modify the machine to have offline learning (where we first train it to remember sequences and then switch off the learning and use purely for prediction) as well as on-line learning. We can enhance the functionality of the machine by having a special symbol to clear the context while keeping the memory intact, and thus make the machine able to recognise multiple sequences rather than one continuous long sequence. We can also have a special character to clear the whole memory if needed. We can also add the functionality to weaken or un-learn a previously learnt association from the memory (because of the nonlinear max algorithm used in learning, it is difficult to erase the previously learnt association completely because subsequent writes to the memory are also affected by a previous written association).

9.3.2 Limitations and weaknesses of the sequence machine

In the implementation with spiking neurons, the way we imposed the restriction that exactly N out of M neurons in each layer should fire was by designing the machine such that in each layer slightly more neurons than N fire in the output, then using feedback inhibition to restrict the output activity to the desired level. One case where this will fail is if the output activity of each layer falls below the required activity of N neurons firing, and the reset inhibition would not know when to stop waiting for the N^{th} output spike from that layer.

Another potential problem is in the case when we are dealing with a layer such as context which takes inputs from multiple layers. **The system has to wait for the required number of inputs from each layer because it cannot distinguish the different input layers for the purpose of counting the input spikes**. If either of its input layers do not fire the required number of spikes, the context layer will keep waiting indefinitely and could classify the first spike of a new burst as belonging to the previous burst, thus spoiling the computations for further layers as well. However, since the context machine is designed to start up at a single input only (a 1-of-A code representing a single letter from the input alphabet) errors such as random spikes are not likely to

occur. The wheel model is linear and so it is possible to precisely tune the behaviour of the system to avoid unwanted errors.

Another weaknesses in our implementation is that **we used the simple linear wheel model of spiking neuron rather than a more biologically plausible model** such as the RDLIF model (which in turn is an implementation of the kernel of the spike response model developed by Gerstner [28]). We had to do so because of our constraint of making the model function exactly as the time-abstracted model (desirable because of our top-down engineering goal that the lower layers must perform to the exact specifications of th higher layers), and we found that it was not feasible to have it reproduce exactly the behaviour of the time-abstracted model using the RDLIF model of spiking neuron (in keeping with our aim of engineering a top-down system, in which the low-level components accomplish the functionality that is specified by the higher level components). In future we can rethink this issue more closely and develop a model that can perform more robustly, even if it does not perform exactly as the time-abstracted model but only approximately.

One weakness of the wheel model is that **it does not have an explicit delay between input and output spikes**, and we would like the neurons to touch the threshold on the activation slope rather than on receiving an input spike (in order to obtain the same relative firing order with the spiking model and the abstracted model). Although we have tried to design the system so that this condition is fulfilled, there may be cases when this will not hold and the order will be lost.

The memory capacity of our model is limited by the choices we have made in our design. Certain other models such as long short term memory [34] are claimed to be quite efficient in remembering past errors in sequence learning and can also do it efficiently. However, this is just a question of changing the training algorithm and the network structure appropriately. The asynchronous update rule framework we have used so far in developing the predictive sequence machine, the criteria we use to judge similarity and the performance measures can still be the same regardless of the learning rule or the structure of the network. We can adapt our on-line asynchronous updating framework (mentioned in chapter 5) to implement other models.

An interesting question regarding the sequence machine is: **if the dimensions of the memory are infinite, what are the computation capabilities of the model**? Will the infinite context have the ability to distinguish very similar sub-sequences, in spite of the ordered N-of-M encoding of the symbols destroying the similarities and magnifying the differences? In the N-of-M SDM, the same address decoders can get activated for two very similar context vectors (because the number of address decoders is small compared to the size of the address space), so the capacity of the memory to distinguish between similar looking contexts is not expected to be good. Such theoretical capabilities of the memory has not been analysed as yet, and the issue is a potential weakness.

The sequence machine has been designed to remember those sequences that are explicitly presented to it. However the ability of the model to accurately predict outputs for sequences that have not been explicitly presented to it for training has not yet been tested, and is another potential weakness.

9.4 Applications of the sequence machine

As mentioned in the discussion of the strengths of the model, the sequence machine can be useful in a variety of applications. The model can be utilised in one of the two ways: either using the sequence machine model in its present form, or tailoring certain components of the model to a specific application.

The sequence machine in its present form, whether implemented using the time-abstracted significance vectors or by using spiking neurons, is useful for applications where we may need to use on-line or one-shot predictive sequence learning (where the system has to learn on a single presentation of the inputs), with gradual rather than abrupt forgetting, in an environment with errors and where an associative memory would be beneficial.

However, the model is unsuitable for applications involving sequences which are generated by a specific grammar or follow a specific pattern, and where the task of the machine is to deduce the grammar and predict future values of a time series after being trained with a number of trials, such as predicting trends in stock market data or robots learning to play baseball or cricket. This is because we did not build the sequence machine assuming any type of generative grammar, and also because it has not been designed to remember higher-level associations or meanings of the symbols involved (although this capability has not yet been

tested, and it may be possible to modify the design of the machine to enhance this functionality). It is designed to remember those sequences that are explicitly presented before it.

In the dissertation, all our experiments involving the sequence machine have been using sequences of alphabets or English characters. However, the sequence machine can be designed to recognise other, more complex sequences such as notes of a tune or images or data chunks. In order to deal with more sophisticated sequences in applications (such as sequences of images), we need to have encoders capable of rotation and translation invariant transformations before a symbol in such a sequence is encoded as an ordered N-of-M vector for use in the sequence machine.

Below we mention some possible applications and some thoughts on how the sequence memory may be used to implement them.

9.4.1 Learning and completion of a sequence of tunes

An interesting application could be to build a machine that is fed a series of songs (each song being learnt as a sequence of tunes). In this application, the symbols constituting the sequence are the tunes of the song. The user hums a few notes of his or her favourite song and the machine can identify to which song the sequence of tunes belongs to, and complete the song. Such an application is possible because of the property in the machine of being able to lock on to a context on being given a few symbols from the middle of a previously learnt sequence, and predict the remaining symbols. However, in order to encode a tune in a proper way, we need to find an appropriate time invariant representation of each tune.

9.4.2 Sequences of characters from any language

If we have a shift register model of sequence machine with a variable register such that the length of time window is equal to the total length of the sequence, it can recognise a sequence of any length. We have already shown that the combined model can perform better than an equivalent shift register. Therefore, given a large enough memory and a large size of context vector, we can claim that the machine would be able to learn and predict any long sequence such as the complete works of Shakespeare. In such a case, we could give it a phrase from the

middle of any of Shakespeare's writings and it should be able to complete that phrase.

A possible application in this field is a prompter (similar to text prompters in mobile phones) that can prompt the completion of words or sentences based on a few input characters. Since the sequence machine also takes the context into account when making a prediction and learns on the go, it could take into account the context of the letters in the word as well as the context of the word in the sentence, when prompting the next letter.

9.4.3 Sequence of images

A movie is a sequence of still images. The sequence machine can be trained to learn a particular sequence of images in a single training epoch, and then predict the next images in a movie using the training set. This could be useful in software rendering in a simple computer game, where we may be required to 'fill in' between image frames which are slow to arrive, and the same scenarios are repeated many times. Learning in a single pass is especially significant here because the images are constantly changing in a real movie and it may be required to learn very fast.

9.4.4 Copying robot gestures

A model of sequence learning developed using a Hopfield net as an auto-associative memory has been used to train a robot to learn and reproduce 50 different types of hand gestures such as the letters of the alphabet [55]. Every gesture was condensed into a sequence of 16 feature vectors that stood for the angle of the hand movement of the robot. It was found that the model could generalise the learnt gestures to previously unlearnt combinations or to variations in the training set.

The correlation matrix memory in the data store layer of the ordered N-of-M SDM that we have used in our model is similar to a Hopfield net (which can also learn in a single pass and can act as a hetero-associative memory) as far as an associative memory and on-line learning is concerned. Therefore, it may be possible to use our model for a similar application. Some interesting applications in this domain would be a robotic traffic controller or a toy robot that can be trained to learn a sequence of simple gestures by a child. However, the ability of our model to generalise unknown sequences has not been properly investigated

yet. Such an investigation and possible extensions to the design of the machine to enhance this ability, is an avenue of further work.

9.4.5 Sequence of phonemes

Another possible application for the sequence machine is learning a sequence of phonemes. NetTalk [67] is a feed-forward neural network trained using back-propagation that was used to associate a letter in a word with its corresponding phoneme sound (a phoneme is the smallest distinguishable phonetic unit in a language). Although the model was not strictly a sequence machine because it could be trained with inverted sequences and the output would still be valid, the solution used similar principles to sequence learning since the pronunciation of any letter depends on its context in the word where its used. Speech can be considered as a sequence of phonemes. As an extension to the NetTalk application, our sequence machine could theoretically be trained to hear any speech or individual sentences and complete them for the speaker (through prediction). Such an application could be useful in helping people with speaking difficulties. Another possible application could be a pre-trained voice prompter in any language, which could be useful in learning a foreign language.

9.4.6 Thorpe's application of SpikeNET

Thorpe's SpikeNET [22] is a concrete example of a model implemented using spiking neurons implementing rank-order codes that is used to recognise a face image in real time from a given database of faces (built using a model of the human visual system). The success of the SpikeNET model shows that it is feasible to build an efficient large-scale system that uses spiking neurons transmitting spikes over multiple layers, and works in an event-driven way, with each spike firing treated as an event in the system. Thorpe's model is different from our sequence machine, however, in that it does not have an associative memory and cannot store what it has learnt separately. It works on a simple model of learning, involving sensitising different neurons to respond to particular input sequences of spikes by keeping the threshold of each neuron so as to be receptive to a specific input rank-order (thus acting as a receptive field). Our model, which uses an associative memory to write associations between symbols, can sensitise a layer of neurons to an input sequence and therefore is more flexible than Thorpe's

model. Therefore, there is scope for adapting it for use in similar applications.

9.5 Extensions to the model and future directions

Given the multidisciplinary scope of this work, there are a number of directions in which the model can be extended and further work can be done. Here we present some suggested avenues where future work may be pursued, including continued research on the core sequence machine and developing extensions.

9.5.1 Better sequence machine

One direction in which this work can be extended is to improve the performance of the sequence machine, either by encoding the context in a more efficient way, or by using models that can train more efficiently. We can explore different architectures for the purpose, while using the same the predictive on-line sequence learning framework.

In developing the sequence machine, we had to make a number of choices about the spiking neural model, the memory architecture, etc. We could explore different parameters and architectures to the model than the ones we have chosen, and study if we can get better memory performance or easier spiking neural or hardware implementation by using those architectures. For example, we could have N-max of M or approximately N-of-M coding instead of strict N-of-M, or use a probabilistic function instead of the max function as the learning rule, or have lateral inputs and negative weights in our model, or a function more similar to the STDP function for connection weight change.

9.5.2 More tests on the sequence machine

There is potential for conducting more tests on the sequence machine, to determine how its performance is effected by different variables from the parameter space. Because of computational constraints, we could not explore the parameter space very thoroughly. We could use these tests to study how to optimise and further improve its performance.

Some of the future experiments can be the following:

Sequences with non-uniform distributions: we can have different distributions for different parts of the sequence, such as parts of the sequence being generated from different alphabets or from different distributions. We can measure the conditions under which the recall of the stored sequence is accurate.

Sequences with varying levels of noise: we can vary the noise in the sequences and measure how the system performs, and what amount of noise can break the system for a specific configuration (context sensitivity parameter, distribution, sequence length, etc).

Measuring the kind of errors in case of wrong recalls: recall errors in the sequence machine can be of various types. For example, the error may be caused due to too much weightage being given to the context as opposed to the input, or vice versa. Errors can be caused due to low activity, with the system being unable to make an informed guess as to what the next prediction should be. Sometimes, the chosen symbol may be incorrect but the output symbol whose activity was second highest or third highest may be the correct symbol. We can list the various types of errors and can measure what is the most common type of errors that we get under different conditions.

Other tests: Other tests could be to determine the highest length of a single sequence that can be recalled correctly and the number of sequences of this length that can be stored, the effective look-back length of symbols over which the machine can distinguish the context, the number of iterations taken by the machine to forget a previously learnt sequence due to a different association being written to the memory, etc.

9.5.3 A more robust spiking neural implementation

In the current spiking neural model, we had to precisely engineer the system to perform correctly. A real world system would not have components that are precisely engineered in this way, so we could take steps to make our model more robust. Some possible ways this can be accomplished are mentioned below.

We could use a larger encoding (bigger value of M in N-of-M), since a system with larger codes is likely to be more stable with respect to temporal dispersion in spike bursts. We could use an alternative neural model with higher order dynamics rather than the wheel model. For example, the RDLIF model with decay is more biologically plausible, and a layer of RDLIF neurons should behave more robustly and stably in response to input spikes, as shown in chapter 6. We could

have an approximate N-of-M code, which could be easier and more realistic to implement with neurons without needing the crude method of feedback reset inhibition. Not having the strict requirement that the spiking neural implementation should perform exactly as the implementation using time-abstracted significance vectors would greatly simplify matters and would also be more realistic. Also, we assume all the spiking neurons of one layer to be homogenous in the present model, i.e. they have the same parameters. It would be interesting to observe if de-homogenising the neurons of a layer, while keeping their parameters within a certain range, makes the whole system more robust to noise.

9.5.4 Different learning rule in spiking neurons

Instead of implementing the max function as the learning rule, we could use a more biologically plausible learning rule such as spike time dependent plasticity [13]. In STDP, the weight change is a function of the time difference between the input and output spikes (as in the standard STDP diagram in the first quadrant). This is more biologically realistic but more complex to implement. Delorme et al [20, 21] have explored modelling with STDP in their SpikeNet simulator.

9.5.5 Avoiding redundancy in the learning rule implementation by spiking neurons

Since the address decoder outputs go to all the neurons of the data store layer, there is redundancy in the memory storage of all the address decoder sensitivities in the data store neurons. A way to avoid redundancy could be to either store the significances of the address decoder outputs in the address decoders themselves, or to store them in a layer of intermediate neurons, each of which connects to exactly one address decoder neuron. Such intermediate neurons can be considered part of the data store layer.

9.5.6 Alternative way to encode significance vectors

An alternate method of encoding the significance vector, instead of using a geometrical progression as we have implemented during this work, could be as a vector of probabilities, where each component represents the probability that the corresponding neuron fires before the neuron after it in the spike burst, where the

firing times are generated by a Poisson process. For example, if the firing order of a burst in a four neuron layer is [A,B,C,D] representing A fires before B, B before C and so on, and the firing times are generated by a Poisson process, then p(A) is the probability that A fires before B in the burst, then the significance value of the component A will be equal to p(A).

9.5.7 Extensions to the combined model

One extension to the combined model could be to make the context sensitivity factor λ dynamic rather then static, enabling us to modulate the relative influence of the context in determining the prediction of the next input. The factor can be changed dynamically based on the confidence of the output prediction or some other suitable mechanism.

Another extension could be to expand the old context and input to a vector of large size in order to make them more linearly separable before adding them, and then adding an extra neural layer to contract the expanded sum of the weighed context and the encoded input to the original size of the context. However, such an extra contract layer following an N-of-M code might reduce the capacity of the memory by destroying information about the old context.

Instead of using N-of-M codes to encode the vectors in the sequence machine, we could use a vector of A elements (A being the size of the alphabet) whose elements encode the relative importance of each symbol in the alphabet, and normalise the vector to have the sum of all elements equal to 1. Such an encoding scheme would make it convenient to identify the relative composition of each input symbol in the context.

9.5.8 Hardware implementation

Implementing a neural system in hardware rather than software has the advantage that components can be processed in parallel using hardware and hence can be faster if large scale neural circuits are used (small circuits would have communication overheads which would reduce the speedup due to the parallelisation). Neural networks, including the sequence machine, have a parallel architecture (since the neurons in a layer are supposed to be processed in parallel) and are therefore suitable for hardware implementation. The spikes could be treated as events and transmitted across different components through common buses. The

neural function can be thought of as a processing element, processed separately from the memory where all the connection weights are stored. Using sparse N-of-M codes to communicate across components consumes low power. The event driven nature of the spiking neural implementation, where computation needs to be performed only when a spike is fired, can also save power and is suitable for large systems if the number of events per unit time is small relative to the size of the system.

The SpiNNaker project at the University of Manchester [68], funded by an EPSRC grant, is exploring scalable hardware implementations of neural networks (aiming to build a large scale neural platform involving millions of neurons that can process calculations in real time and work much faster than a comparable software implementation), working on similar principles of having different components in the system communicate through spike events. It could be a possible hardware platform to test many of the ideas presented in this dissertation.

9.5.9 Modelling parts of the brain

Although the sequence machine model is not biologically accurate in its present form, it can be adapted to implement parts or areas of the brain having similar structure or function, in both hardware and software, with minor modifications. In chapter 3 we had studied different biological systems related to sequence learning. The sequence machine could potentially be adapted to model many of those.

As mentioned earlier, this model is a concrete implementation of the speculative but nevertheless interesting prediction model of the neocortex as proposed by Jeff Hawkins [31]. The model of the neocortex as per the predictive learning framework presented in his book, is functionally similar (although not structurally because we do not use hierarchical structures as proposed in that book) to the way the sequence machine learns associations and predicts the next symbols in a given sequence on-line. A modified version of the sequence machine can thus lead to a hardware or software implementation of neocortical function according to that model.

Our model also has both short-term and long-term memory components. It can be explored whether it can implement some component of a model of working memory in humans as proposed by Baddeley [8].

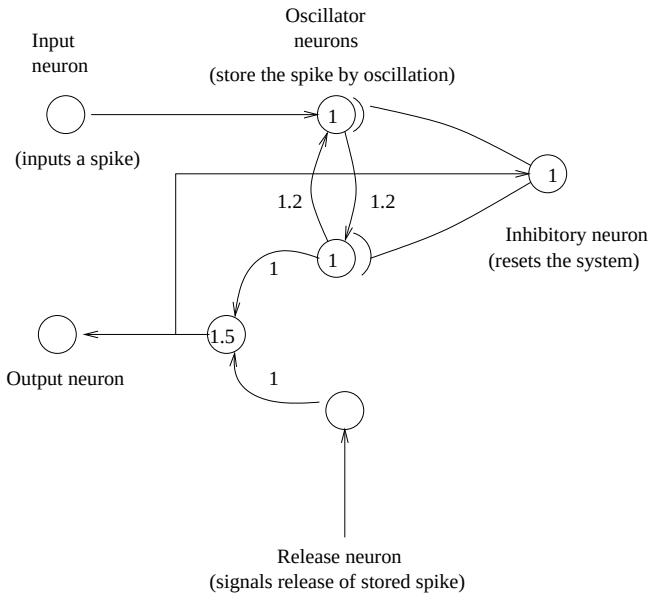

Figure 9.1: A coupled oscillator network to store a spike and release it on getting another spike. The numbers in the circles denote neural thresholds, and the connection weights are shown beside the arrows connecting different neurons

9.5.10 Alternative ways to implement the spike timing constraints

We could have alternative ways to synchronise the firing times in different layers of neurons, as mentioned below.

Adding axonal delays

One method to ensure that spike firings are synchronised is to fine-tune the neural parameters such as axonal delays. Axonal delays are delays in the transmission of a spike through the axon of a neuron. In effect, they translate to a fixed temporal delay between a neuron firing and the sensing of that spike at the inputs of the next layer. We can design the system by engineering the axonal delays (or inserting extra delay neurons if necessary) to ensure that the timing relations between different spike bursts are maintained.

Latching a spike

One way to synchronise firings could be to freeze or latch the spikes of a layer and release them in response to spikes acting as control signals, similar to latches

used in standard asynchronous circuit design [71]. However, latching a spike is a problem in neural systems, because we assume the spike to be generated immediately (or with a fixed axonal delay) as the neural activation exceeds the threshold.

A way to implement a neural latch, that can store a spike and release it on receiving a signal, could be by using a coupled oscillator network, consisting of a pair of neurons connected to each other, as shown in figure 9.1. An input spike to the first neuron is transmitted to the second and from the second to the first. The spike to be stored is input to the first neuron of the pair through an unidirectional connection, where it keeps oscillating between the two neurons and is thus 'stored' until the system gets a 'release' spike from a third neuron. A fourth neuron is fed input spikes from one of the pair of oscillatory neurons as well as this third neuron, and fires a spike only if both spikes occur (which can be possible, for example, if its threshold is 1.5 and the connection weights from both these neurons are equal to 1). We can assume that the spiking model used is the standard leaky integrate and fire model [53], so if a neuron does not get a spike for sometime its activation decays to a resting value, and if the two input spikes to the fourth neuron are not coincident the neuron can integrate the spikes and fire when the activation exceeds its threshold. When the fourth neuron fires and 'releases' the stored spike, both the oscillatory neurons can be reset through a fifth resetting neuron. Alternatively, the third 'release' neuron can itself be connected to the pair of oscillating neurons and resets them, or can itself act as a gate (similar to gated ion channels in biological neurons) to shut down the oscillatory pathway and open the pathway to the output, without the need of a fourth neuron. We have not actually implemented this system, but the above explanation is illustrative of how it might be used.

The problem with such a coupled arrangement to store a spike is that we need also to maintain the ranks of the spikes in the burst (in addition to freezing a burst and releasing it on getting a signal) as this forms the neural code being transmitted. A coupled oscillator network as explained above can lose the relative rank because the phases of the oscillating neurons and the time of firing of the released spike in such a network cannot be predicted, although we can guarantee that they will fire some time after receiving the control signal to release the spike.

One way we can freeze the relative timings of a burst of spikes emitted by a layer and release them on getting a release signal is by using a dynamic threshold.

As per the rank-order code, we need to maintain the order of spikes in the burst. We can keep the threshold initially to a high level so none of the neurons in the layer fire, yet the order of their activations is maintained. When we need to release the spikes of that layer, we gradually decrease the threshold of all the neurons, which will lead to them firing one by one in the order of their activation, the neuron with the highest activation firing first (because it reaches the steadily decreasing threshold earliest).

9.5.11 Similarities with asynchronous logic

This work has many similarities to standard asynchronous logic design [71], especially to a family of asynchronous logic circuits called self-resetting logic, and it may be worth considering the similarities and their implications.

In asynchronous logic design, communication takes place by transmission of electrical signals through wires, which can be considered similar to transmission of spikes in our neural model. The electrical signals transmitted are in one of the two levels 0 or 1 (following binary logic), and the switching of levels could be considered as an event similar to firing a spike. A standard asynchronous logic circuit has the handshake as its defining constituent to synchronise timing between components interacting with each other and passing data signals. If we consider groups of spiking neurons interacting with each other, they show interactions similar to handshaking, in the sense that they can excite each other to generate corresponding bursts of spikes, which may be thought as the 'request' and 'acknowledge' signals. Also, we have considered in chapter 6 how a neural circuit could implement a latch, which is a standard component in asynchronous logic design that holds a signal and releases it on receiving a control signal. Such a circuit could be designed to hold a spike indefinitely and release it on getting a 'request' spike from another neuron. The released spike can be considered as the 'acknowledge' signal in response to the 'request' signal.

Apart from the above mentioned similarities of asynchronous logic circuits with spiking neural systems in general, the sequence machine that we implemented using spiking neurons is essentially asynchronous in nature, since there is no global synchronising mechanism such as a clock. In the development of the system using spiking neurons, we have dealt with issues common to asynchronous circuits such as timing issues involving synchronising different autonomous components without the use of a global clock.

If we can show how each component in traditional asynchronous design, such as the latch and the handshake protocol, can be implemented in neurons, and adapt traditional asynchronous design components such as latches to take into account the additional constraints of spiking neural implementation, we could potentially reach a stage where it is possible to translate any spiking neural network into its equivalent asynchronous logic circuit. There are a number of tools for timing analysis and synthesis of asynchronous circuits available in asynchronous design that can potentially be adapted for engineering or analysing spiking neural circuits. For example, there are synthesis tools such as Balsa [71] that can perform automated synthesis from a high level description of the functionality of the system (in a suitable description language) to low level implementation using asynchronous components. In the neural case, at some stage it might be possible to have a similar tool to code the high level functionality of the system (such as a sequence machine) in a suitable way and automate the synthesis of the system using spiking neurons, with their connections and other parameters optimised automatically.

9.6 Significance of the research and deeper implications

To summarise, this dissertation describes a constructionist way to develop insights into how a high-level system can be built using low-level spiking neurons as components. Our primary achievements are in designing a working model that successfully accomplishes a high-level task using spiking neurons, and highlighting the problems and issues that arise at different stages of the modelling. The solutions that we have proposed to many of the problems may not be ideal, but we hope they will stimulate interested researchers to find better solutions.

Since our framework for on-line predictive learning is independent of the actual implementation of the learning rule, it can theoretically be used for any algorithm or architecture with the same aim. Also, issues related to the dynamics of neural interactions (such as the stability and coherence of spike bursts), timing relations (such as the synchronisation of different bursts) and learning (such as storing order information using eligibility vectors) are independent of the spiking neuron model used.

Rather than being just an implementation of a sequence machine in spiking

neurons, this work could be seen in a wider context of developing large real-time systems to perform useful tasks using spiking neurons as components. In our modelling of the sequence machine, we have highlighted some problems in our task and proposed solutions. Although our solutions may be specific to the task, many problems we have highlighted are universal to any system trying to perform a high-level task using low-level components. Therefore, the insights from our work should be useful for modellers of biological neural systems or of complex or emergent phenomena in general.

This work can be considered in the context of developing spiking neural implementations of different kinds of high-level systems (not only the sequence machine). It can be considered the first step in the direction of building a computing paradigm that utilises the connectionist structure of neural models in building distributed, error-tolerant and robust systems, in both hardware and software, which can be utilised to perform higher-level computing functions.

Bibliography

[1] M. Abeles. *Corticonics: neural circuits of the cerebral cortex*. Cambridge University Press, 1991.

[2] E. D. Adrian. The impulses produced by sensory nerve endings. *Journal of physiology*, 61, 1926.

[3] S. American, editor. *Scientific American Book of the Brain*. The Lyons Press, 1999.

[4] B. Ans, Y. Cotton, J. C. Gilhodes, and J. Velay. A neural network model for temporal sequence learning and motor programming. *Neural Networks*, 7(9), 1994.

[5] M. Arbib, editor. *The Handbook of Brain Theory and Neural Networks*. MIT Press, 2003.

[6] J. Austin. Adam: A distributed associative memory for scene analysis. *Proceedings of First International Conference on Neural Networks Ed. M Caudhill and C Butler, San Diego*, 4, June 1987.

[7] J. Austin and M. Turner. Matching performance of binary correlation matrix memories. *Neural Networks*, 10, 1997.

[8] A. Baddeley. *Working Memory*. Oxford University Press, 1986.

[9] L. E. Baum. An inequality and associated maximization technique in statistical estimation for probabilistic functions of a Markov process. *Inequalities*, 3, 1972.

[10] R. Beale and T. Jackson. *Neural computing: An introduction*. IOP Publishing Ltd, 1990.

[11] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.

[12] L. Berthouze and A. Tijsseling. A neural model for context-dependent sequence learning. *Neural Processing Letters*, 1, 2006.

[13] G. Q. Bi and M. M. Poo. Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. *The Journal of Neuroscience*, 18, 1998.

[14] J. M. Bower and D. Beeman. *The Book of Genesis: Exploring Realistic Neural Models With the General Neural Simulation System*. Springer, 1998.

[15] N. Carnevale and M. Hines. *The NEURON Book*. Cambridge University Press, 2006.

[16] W. Cathy, M. Berry, S. Shivakumar, and J. McLarty. Neural networks for full-scale protein sequence classification: Sequence encoding with singular value decomposition. *Machine Learning*, 21, 1995.

[17] M. Cumpstey. *SpikeNetwork: a spiking neural simulator*. Personal communication, 2006.

[18] H. H. Dale. Pharmacology and nerve-endings. *Proceedings of the Royal Society of Medicine*, 28, 1935.

[19] P. Dayan and L. Abbot. *Theoretical neuroscience*. MIT Press, 2001.

[20] A. Delorme and L. Perrinet. Network of integrate-and-fire neurons using rank order coding a: how to implement spike timing dependant plasticity. *Neurocomputing*, 38, 2001.

[21] A. Delorme and L. Perrinet. Network of integrate-and-fire neurons using rank order coding b: spike timing dependant plasticity and emergence of orientation selectivity. *Neurocomputing*, 38, 2001.

[22] A. Delorme and S. Thorpe. Spikenet: an event driven simulation package for modeling large networks of spiking neurons. *Neural Networks*, 14, 2003.

[23] J. L. Elman. Finding structure in time. *Cognitive Science*, 14, 1990.

[24] G. B. Ermentrout and J. Rinzel. Beyond a pacemaker's entrainment limit: phase walk-through. *Regulatory, Integrative and Comparative Physiology*, 246, 1984.

[25] J. Feng. *Computational Neuroscience: A Comprehensive Approach*. Chapman and Hall/ CRC Press, 2004.

[26] S. Furber, G. Brown, J. Bose, M. Cumpstey, P. Marshall, and J. Shapiro. Sparse distributed memory using rank order neural codes. *Accepted in IEEE Transactions on neural networks*, 2006.

[27] S. Furber, J. Cumpstey, W. Bainbridge, and S. Temple. Sparse distributed memory using N-of-M codes. *Neural Networks*, 10, 2004.

[28] W. Gerstner and W. M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 1st edition, 2002.

[29] I. I. Guyon, L. Personnaz, J. P. Nadal, and G. Dreyfus. Storage and retrieval of complex sequences in neural networks. *Physical Review*, 38, 1988.

[30] B. Hammer and P. Tino. Recurrent neural nets with small weights implement definite memory machines. *Neural Comp.*, 15, 2003.

[31] J. Hawkins and S. Blakeslee. *On intelligence*. Henry Holt and Company, New York, 2004.

[32] S. Haykin. *Neural Networks: a Comprehensive Foundation (2nd Ed)*. Prentice Hall, 1999.

[33] D. O. Hebb. The organisation of behaviour: a neuropsychological theory. *Bulletin of Mathematical Biophysics*, 1949.

[34] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9, 1997.

[35] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences, USA*, 79, 1982.

[36] E. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14, 2005.

[37] E. M. Izhikevich. Polychronization: Computation with spikes. *Neural Computation*, 18, 2006.

[38] H. Jaeger. The echo state approach to analysing and training recurrent neural networks. Technical Report 148, German National Research Center for Information Technology, 2001.

[39] A. Jagota, G. Narasimhan, and K. W. Regan. Information capacity of binary weights associative memories. *Neurocomputing*, 19, 1998.

[40] E. C. Jennings and S. Aamodt. *Nature neuroscience: Computational approaches to brain function.* Nature Publishing Group, 2000.

[41] D. Jin. Spiking neural network for recognising spatiotemporal sequences of spikes. *Physical review*, 69, 2004.

[42] R. S. Johansson and I. Birznieks. First spikes in ensembles of human tactile afferents code complex spatial fingertip events. *Nature Neuroscience*, 7, 2004.

[43] M. I. Jordan. Serial order: a parallel distributed processing approach. Technical Report 86044, Institute for Cognitive Science, University of California San Diego, 1986.

[44] E. R. Kandel, J. H. Schwartz, and T. M. Jessell. *Principles of Neural Science.* McGraw Hill, 4 edition, 2000.

[45] P. Kanerva. *Sparse Distributed Memory.* MIT Press, 1988.

[46] D. Kim. A spiking neuron model for synchronous flashing of fireflies. *BioSystems*, 76, 2004.

[47] A. Knoblauch. *Synchronization and pattern separation in spiking associative memories and visual cortical areas.* PhD thesis, University of Ulm, Germany, 2003.

[48] K. Koch and I. Segev, editors. *Methods in neuronal modelling: From ions to networks.* MIT Press, 1998.

[49] T. Kohonen. Correlation matrix memories. *IEEE Transactions on Computers*, 21(4), April 1972.

[50] D. Kustrin and J. Austin. Spiking correlation matrix memory. *Citeseer (researchindex): citeseer.ist.psu.edu/315059.html*, 1, 1999.

[51] K. J. Lang and G. E. Hinton. The development of the time delay neural network architecture for speech recognition. Technical Report 88152, Carnegie Mellon University, 1988.

[52] A. Longstaff. *Instant Notes in Neuroscience*. Garland Group, Taylor and Francis, 2000.

[53] W. Maass and C. M. Bishop. *Pulsed Neural Networks*. MIT Press, 1999.

[54] W. Maass, T. Natschlger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14, 2002.

[55] A. Maurer and A. Billard. Extended hopfield network for sequence learning: Application to gesture recognition. *Proceedings of the ICANN 2005, Warsaw, Poland*, 1, 2005.

[56] V. Mountcastle. *An Organizing Principle for Cerebral Function: The Unit Model and the Distributed System, in The Mindful Brain (Gerald M. Edelman and Vernon B. Mountcastle, eds.)*. MIT Press, 1978.

[57] M. C. Mozer. *Neural network architectures for temporal sequence processing, in Predicting the future and understanding the past*. Addison Wesley, 1993.

[58] M. C. Mozer. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multiscale processing. *Connection Science*, 6, 1994.

[59] W. X. Pan, R. Schmidt, J. R. Wickens, and B. I. Hyland. Dopamine cells respond to predicted events during classical conditioning: Evidence for eligibility traces in the reward-learning network. *Journal of Neuroscience*, 25, 2005.

[60] S. Panzeri, R. Petersen, S. R. Schultz, M. Lebedev, and M. E. Diamond. The role of spike timing in the coding of stimulus location in rat somatosensory cortex. *Neuron*, 29, 2001.

[61] I. P. Pavlov. *Conditioned Reflexes: An Investigation of the Physiological Activity of the Cerebral Cortex (translated by G. V. Anrep).* Oxford University Press, 1927.

[62] T. A. Plate. Holographic reduced representations. *IEEE Transactions on Neural Networks*, 6, 1995.

[63] W. Rall. Theory of physiological properties of dendrites. *Annals of New York Academy of Sciences*, 96, 1962.

[64] A. S. Reber. Implicit learning of artificial grammars. *Journal of Verbal Learning and Verbal Behavior*, 5, 1967.

[65] F. Rieke, D. Warland, R. de Ruyter van Steveninck, and W. Bialek. *Spikes: Exploring the neural code.* MIT Press, 1999.

[66] F. Schwenker, F. T. Sommer, and G. Palm. Iterative retrieval of sparsely coded associative memory patterns. *Neural Networks*, 9, 1996.

[67] T. J. Sejnowski and C. R. Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, 1, 1987.

[68] S.Furber, S.Temple, and A.Brown. On-chip and inter-chip networks for modelling large-scale neural systems. *Proceedings of International Symposium on Circuits and Systems, ISCAS 06*, 1, 2006.

[69] L. Shastri. A computational model of episodic memory formation in the hippocampal system. *Neurocomputing*, 38, 2001.

[70] E. D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems. (2nd Ed).* Springer, New York, 1998.

[71] E. J. Sparso and S. Furber. *Principles of Asynchronous Circuit Design, a systems perspective.* Kluwer Academic Publishers, 2001.

[72] D. C. Sterratt. *Spikes, synchrony, sequences and Schistocerca's sense of smell.* PhD thesis, University of Edinburgh, UK, 2002.

[73] M. Strickert, B. Hammer, and S. Blohm. Unsupervised recursive sequence processing. *Neurocomputing*, 63, 2005.

[74] R. Sun and C. L. Giles, editors. *Sequence Learning.* Springer-Verlag, 2000.

[75] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1997.

[76] S. Thorpe, A. Delorme, and R. VanRullen. Spike based strategies for rapid processing. *Neural Networks*, 14(6), 2001.

[77] S. Thorpe, D. Fize, and C. Marlot. Speed of processing in the human visual system. *Nature*, 381, 1996.

[78] P. Tino and G. Dorffner. Building predictive models from fractal representation of symbolic sequences. *Proceedings of Neural Information Processing Systems (NIPS)*, 12, 2000.

[79] T. P. Trappenberg. *Fundamentals of computational neuroscience*. Oxford University Press, 2002.

[80] T. Troyer and A. Doupe. An associational model of birdsong sensorimotor learning: efference copy and the learning of song syllables. *Journal of Neurophysiology*, 84, 2000.

[81] T. W. Troyer and A. J. Doupe. An associational model of birdsong sensorimotor learning 2: temporal hierarchies and the learning of song sequence. *Journal of Neurophysiology*, 84, 2000.

[82] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8, 1992.

[83] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, 1974.

[84] D. J. Willshaw, O. P. Buneman, and H. C. Longuet-Higgins. Non holographic associative memory. *Nature*, June 1969.

[85] H. Wilson. *Spikes, Decisions, and Actions - The Dynamical Foundations of Neuroscience*. Oxford University Press, 1999.

[86] F. Worgotter and B. Porr. Temporal sequence learning, prediction and control - a review of different models and their relation to biological mechanisms. *Neural Computation*, 17, 2005.

Appendix A

Implementing the temporal abstraction

A.1 Introduction

We want to examine if it is possible to implement the temporal rank order abstraction (encoding a symbol, made of a sequence of spike firings in a layer of neurons, as a vector of significances) using the rate based leaky integrate and fire (RDLIF) model of spiking neuron as described in chapter 6.

We consider a network of two layers with two neurons in each layer, as shown in figure 1. Layer 1 has neurons 1 and 2 and layer 2 has neurons 3 and 4. The two layers are connected with their respective connection weights w_{31}, w_{32}, w_{41} and w_{42} as shown in figure A.1.

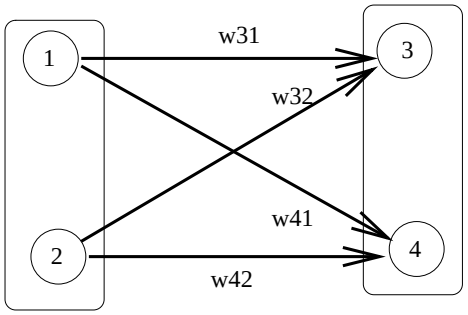

Figure A.1: A simple 2-layer network

A.1.1 Problem statement

It is given that neuron 1 fires (at time t_1) before neuron 2 (at time t_2). It is also given that the parameters of the system (such as neural connection weights and significance ratios) are such that according to the time abstracted model, neuron 3 is calculated to fire before neuron 4. We want to determine the condition in which neuron 3 will fire before neuron 4 in the equivalent RDLIF model using the same parameters.

A.1.2 Abstracted model

As per the rank order abstracted model, let s_1 be the initial significance and α is the significance ratio common to both the neural layers. For example, when the first input to a neuron fires its effect is weighed by a significance of s_1, and the second input by a significance of $s_2 = \alpha s_1$.

A.1.3 Simplified RDLIF model

We use a simplified RDLIF model with no decay of activation or activation rate. Every time a neuron gets an input spike, its activation is linearly increased by a slope that which is the product of the connection weight and sensitivity to the input spike (which is a function of the order of the input spike). For simplification we assume that the slope stays constant until an input fires, in which case either the slope gets incremented, or the neuron itself fires and is reset, or else the whole neural layer is reset. We can see how the activation of the neurons changes with time in figure A.2.

A.1.4 Calculation

Since it is given that according to the time abstracted model, neuron 3 fires before neuron 4, the calculated activation of neuron 3 would have to be more than that of 4.

$$w_{31}s_1 + w_{32}s_2 > w_{41}s_1 + w_{42}s_2 \tag{A.1}$$

Now, in the simplified RDLIF model, the activation a_3 and a_4 of the neurons 3 and 4 as a function of time t is given as follows:

$$a_3(t) = w_{31}s_1(t - t_1) + w_{32}s_2(t - t_2) \tag{A.2}$$

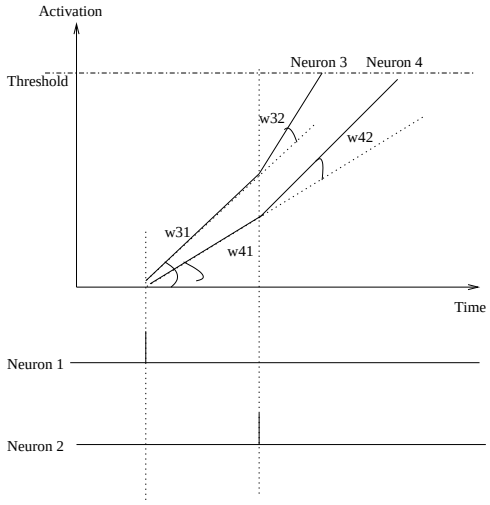

Figure A.2: Simplified RDLIF model of the network

$$a_4(t) = w_{41}s_1(t - t_1) + w_{42}s_2(t - t_2) \tag{A.3}$$

Now if neuron 3 fires at time t_3, its activation at that time $a_3(t_3)$ will be equal to the threshold Θ.

$$\Theta = a_3(t_3) = w_{31}s_1(t_3 - t_1) + w_{32}s_2(t_3 - t_2) \tag{A.4}$$

Solving for t_3 we get:

$$t_3 = \frac{\Theta + w_{31}s_1t_1 + w_{32}s_2t_2}{w_{31}s_1 + w_{32}s_2} \tag{A.5}$$

Similarly for neuron 4, which fires at time t_4, solving for t_4 we have:

$$t_4 = \frac{\Theta + w_{41}s_1t_1 + w_{42}s_2t_2}{w_{41}s_1 + w_{42}s_2} \tag{A.6}$$

The condition for $t_3 < t_4$ is the following:

$$\frac{\Theta + w_{31}s_1t_1 + w_{32}s_2t_2}{w_{31}s_1 + w_{32}s_2} < \frac{\Theta + w_{41}s_1t_1 + w_{42}s_2t_2}{w_{41}s_1 + w_{42}s_2} \tag{A.7}$$

Now, we know that $t_1 < t_2$ since it is given that neuron 1 fires earlier than neuron 2. So let $t_2 = t_1 + k$ where k is a positive value. Therefore we have

$$\frac{\Theta + w_{31}s_1t_1 + w_{32}s_2(t_1 + k)}{w_{31}s_1 + w_{32}s_2} < \frac{\Theta + w_{41}s_1t_1 + w_{42}s_2(t_1 + k)}{w_{41}s_1 + w_{42}s_2} \tag{A.8}$$

Simplifying and removing common terms, we get:

$$\frac{\Theta}{w_{31}s_1 + w_{32}s_2} + \frac{w_{31}s_1 + w_{32}s_2}{w_{31}s_1 + w_{32}s_2}t_1 + \frac{k}{w_{31}s_1 + w_{32}s_2} < \frac{\Theta}{w_{41}s_1 + w_{42}s_2} + \frac{w_{41}s_1 + w_{42}s_2}{w_{41}s_1 + w_{42}s_2}t_1 + \frac{k}{w_{41}s_1 + w_{42}s_2} \tag{A.9}$$

Cancelling t_1 on both sides, we have:

$$\frac{\Theta + k}{w_{31}s_1 + w_{32}s_2} < \frac{\Theta + k}{w_{41}s_1 + w_{42}s_2} \tag{A.10}$$

The above equation has the term $\Theta + k$ common in the numerator, so we cancel it out reversing the inequality sign. Also, since the sensitivity factor α is multiplicative (as per our model), we have $s_2 = \alpha s_1$. Therefore, substituting and cancelling s_1 on both sides, we have:

$$w_{31} + w_{32}\alpha > w_{41} + w_{42}\alpha \tag{A.11}$$

The above equation cannot be simplified any further. We can see that we are unable to cancel out the terms α (the significance factor or sensitivity factor) or the connection weights (w_{31},w_{32},w_{41},w_{42}, which are the slopes in the RDLIF model).

$$f(w_{31}, w_{32}, w_{41}, w_{42}, \alpha) > 0 \tag{A.12}$$

A.2 Conclusion

We have shown that the condition for a simple 2-neuron network following the RDLIF neural model to behave equivalently as a similar network using the time abstracted significance vectors is not independent of the connection weights and the significance ratio. It is not feasible for us to ensure that such a condition is met for every pair of neurons. Therefore, we argue that implementing the temporal abstraction by the RDLIF model of spiking neuron is not feasible in the general case.

Appendix B

Using the SpikeNetwork neural simulator

B.1 The Simulator

The SpikeNetwork spiking neural simulator[17] uses a combination of the event queue model along with a timestep model for a small time window of firing. The simulator is quite modular and flexible, and written in an object oriented way, so it is fairly easy to incorporate new neural models, both simple and complex as long as the interfaces with the other parts of the code remain the same. The model can incorporate features such as synaptic memory and can be used with different spiking neural models suitable for implementing the sequence machine such as rate driven leaky integrate and fire (RDLIF model) and wheel (or firefly) model.

The simulator takes as input a network configuration file and a simulation file. The simulation file starts off the simulation by specifying the firing times of input spikes. The network configuration file takes in the specifications of different types of neurons in the system (which can follow the same model with same or different parameters such as wheel model, or can follow different models such as one neuron following leaky integrate and fire and another neuron following the wheel model), neural layers, connection information between different neural layers, input connection weight matrices of different neural layers and some added layer specific parameters such as significance ratio, and neuron sensitive parameters. The output file gives a series of firing times of neurons in different layers.

B.2 Using the simulator

B.2.1 Input and simulation file format

An example of a typical input file to the simulator is given below to illustrate the format. It represents a network of two neural layers (input and encoder) with 3 wheel neurons in each layer.

```
neuron wheel {
    actrate  0.1;
    threshold  3.0;
    threshtau  100000000;
    threshfix  0.00;
    refractory  0.00;
    significance 0.97;
    sigdec  0.0;
    ffsidata     0.97;
    inputstep  1.00;
} fast;

fascicle medium {
    fast 3;
} input,encoder;

encoder.connect(input);

input.nofm = 1;
encoder.nofm = 3;
input.ffsiinput = 0.97;
encoder.ffsiinput = 0.97;

encoder.weight 0 dense 0.97000 1.00000 0.94090;
encoder.weight 1 dense 0.94090 0.97000 1.00000;
encoder.weight 2 dense 1.00000 0.94090 0.97000;
```

The simulation file format is given below. The first line specifies the time step size and how long should the simulation run. The following lines specify the

initial spikes with their time of spiking.

```
simulate(0,35.0,0.2);
input.spike(7,0.0);
```

B.2.2 Running the simulator

The command to run the simulator is: java SpikeNetwork -pg -nnetwork_file -fsimulation_file

B.2.3 Format of the output file

An example of the output of the simulator is as follows:

```
 0.0000 input:7
50.0000 encoder:11
50.3000 encoder:6
50.5910 encoder:15
```

B.3 Plotting the outputs

Plotting the outputs is done by another Perl script which translates the neural firing times in the simulator output from a lookup table into a form more suitable for processing (with absolute numbers rather than names of the neural layers). The output of this Perl script is, in turn, fed to a Matlab code which then plots the firing times of neurons different layers with respect to time, so we can clearly see the relations between different firing times and how a wave of spikes propagates across different layers in the system.