Liquid State Machine and Its Applicability
to Recognition Problems

BSc (Hons) Computer Information Systems

Zebin Jiang

May, 2005
Liquid State Machine and Its Applicability for Recognition Problems

Submitted by Zebin Jiang

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/intelprop).

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed: ..........................

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed: ..........................
Acknowledgement

With many thanks to my supervisor, Dr Dan Richardson, for his never-ending encouragement and patience through the project; introducing the concept of neural network and liquid state machine

I am grateful for the suggestions made by my friends: Fred Tang, Yanyu Jiang, Ling Cao, Lucy Liu, Joe Yin, Damu Li, and Rong Hu

I would also like to express my thanks to my girlfriend, Ying E, for her support and tireless dedication.

Finally, as always, my gratitude goes to my parents, Yuan-chun Jiang and Hong-xia Yang
Abstract

This project investigated a recently proposed model Liquid State Machine and its applicability for recognition problems in real life. After first introducing some background knowledge about pattern recognition, spiking neuron models which are used as liquid filers in the LSM model are presented in brief. They can pre-process time series of spikes before the readout units can easily extract information from recent input.

With a series of experiments in CSIM, we show the different performances this model has by employing different learning algorithms including linear regression, backpropagation, linear classification and parallel delta rule. An important problem in regard to generating LSM is what the appropriate parameters could be set to influence the performance of this model. Several examples are given to show the various performances by changing those parameters. This information should be of value to others deciding what the best setup of this model is to get a higher performance. To conclude, we briefly touch on the application of LSM in practice with the intention of facilitate future work on this subject.
Index

Chapter 1  Introduction ................................................................................................7
Chapter 2  Pattern Recognition ..................................................................................10
  2.1  Temporal Pattern Recognition ...........................................................................11
      2.1.1  Time-Warp Invariant Speech Recognition ..............................................11
      2.1.2  Analog-Match Problem ...........................................................................12
  2.2  Hidden Markov Models .....................................................................................12
      2.2.1  What Is a Hidden Markov Model ...........................................................12
  2.3  Recurrent Neural Networks .............................................................................14
Chapter 3  Spiking Neural Networks .......................................................................16
  3.1  Hodgkin-Huxley Model ...................................................................................17
  3.2  Integrate-and-Fire Neurons ............................................................................18
  3.3  Synapses .........................................................................................................19
      3.3.1  Synaptic plasticity ....................................................................................20
Chapter 4  Liquid State Machines ...........................................................................22
  4.1  Formal Definition .............................................................................................22
      4.1.1  Existing Similar Approach ......................................................................23
  4.2  Computational Power .......................................................................................24
      4.2.1  Separation and Approximation Property ................................................24
Chapter 5  Learning Algorithms ..............................................................................26
  5.1  Parallel Delta Rule ............................................................................................26
      5.1.1  Derivation of Delta Rule .........................................................................27
  5.2  Backpropagation ..............................................................................................28
      5.2.1  Levenberg-Marquardt (trainlm) ...............................................................29
  5.3  Hebbian learning ..............................................................................................30
Chapter 6  Noise ........................................................................................................31
Chapter 7  Generating Liquid State Machines .......................................................32
  7.1  Environment .....................................................................................................32
      7.1.1  CSIM .......................................................................................................32
      7.1.2  Circuit-Tool .............................................................................................32
      7.1.3  Learning-Tool ..........................................................................................32
  7.2  Other Issues ......................................................................................................33
      7.2.1  Simulation ...............................................................................................33
      7.2.2  Input and Output .....................................................................................34
Chapter 8  Experimental Design ..............................................................................35
  8.1  Overview ..........................................................................................................35
  8.2  Experiment Objectives ......................................................................................35
  8.3  Experiment Key ................................................................................................35
  8.4  Experiment Scenario ........................................................................................36
Chapter 9  Results and Analysis ..............................................................................38
  9.1  Experiment 1: Comparison of Learning Algorithms ........................................38
      9.1.1  Method .....................................................................................................38
      9.1.2  Results and Analysis ..............................................................................39
  9.2  Experiment 2: Noise Sensitivity ....................................................................42
      9.2.1  Method .....................................................................................................42
      9.2.2  Results .....................................................................................................42
Chapter 10 Conclusion and Future Application ....................................................43
Introduction

The Liquid State Machine model is a new approach that theoretically allows for analysis of the real-time computing with continuous input time series. LSM is a spiking neural network which does not require a task-dependent construction of a neural network. Theoretically spiking neurons are computationally powerful as they are able to react non-linearly to individually timed inputs. Consequently, the Liquid State Machine is a very powerful model. The LSM model comprises three parts, an input layer, a large randomly connected core which has the intermediate states transformed from input, and an output layer. Given a time series of input, the machine can produce a time series of behaviour. To get the desired behaviour, we will have to adjust the weights on the links between the core and the output. According to the theoretical analysis of this mathematical model of liquid computer, there are no limitations for its power. So, if we believe this, we would expect that anything that humans or animals can do in real time can also be done by such a machine, with appropriate weights on the links between core and output.

Neural networks have found success in the area of pattern recognition. Although static pattern recognition is already quite difficult from a computational point of view, detecting patterns over time is a complete different story. Most good pattern recognition techniques available today are confined to snapshot recognition; they literally attempt to recognize a pattern in one time-slice of input space while LSM wants to try to recognize a pattern in an input sequence.

There are quite a number of ways of detecting repeating features in single snapshots, but there are very few methods that do the same equally well for patterns over time. Even fewer can handle with noisy data. As the world around us is particularly noisy, for real-world pattern recognition we need methods that are able to cope with the inherent noisy data that comes from it. Biological systems deal with noise very well and so do systems that mimic nature like artificial neural networks. However, there are serious problems with these computational techniques when it comes to training and controlling their dynamics when dealing with input over time.

1.1.1 Outline

Firstly, we provide a brief introduction of pattern recognition and neural network especially the spiking neural network model. We also give an overview of existing techniques hidden Markov models and recurrent neural networks. All these techniques have their individual strengths and weaknesses for application on this particularly complex task. We will pay extra attention to the feature that allows for fair comparison with the Liquid State Machine framework.

Chapter 4 introduces the formal definition of LSM which then bring us to the generation of Liquid State Machine. The “generating liquid state machine” will help
us to carry out the experiments. After analyzing the results of the experiments, we draw a final conclusion.
Chapter 2  Pattern Recognition

This section will introduce some basic background knowledge about recognition. Firstly, we will start with defining what a pattern is. By instinct it can be defined as a reasonably reliable set of characteristics of a thing, group, abstract feature or concept. This makes that the act of pattern recognition consists of identifying such an object by some or all of the features in the information we have about it. Formally, pattern recognition, a branch of artificial intelligence, that can be defined as a process which identifies an object based on analysis of its features. It is currently the dominating field for the application of neural networks. A pattern recognition system can be considered as a two stage device: [1]

a) **Feature Extraction:** at this stage, a feature is defined as a measurement token on the input pattern that is to be classified. Typically, features that will provide a definite characteristic of that input type will be sought. For example, in a text recognition process, if we wish to distinguish the letter “F” from the letter “E”, we would need to compare the number of horizontal strokes in the character. (See figure 1)

b) **Classification:** once the measured features are collected, they will be passed to the classifier which will map these features onto a classification state. At this stage, the classifier will have to decide which type of class category they match closely.

![Figure 1 E's can be distinguished from F's by the fact that E has more horizontal strokes than F](image)

The difficult part of pattern recognition is the ‘characteristics’ can be anything in any form, shape and values of that data we’re dealing with. Also, there are many types of data, ranging from well constrained numerical information gathered via interviews, questionnaires or surveys to raw sensory input acquired directly from the real world. As often is the case, less is more. In general, having a lot of characteristics in the data makes things worse; it will be harder for people to recognize those stable features,
which will assist them to identify patterns reliably

2.1 Temporal Pattern Recognition

As pattern recognition is becoming more and more important, it has been developed into a large area of computer science. A lot of experiments and research have shown that static object recognition is relatively easy to be carried out by computers. However, in real life, it is difficult to find stable and consent objects, particularly when the objective is to detect some event or action, process it with a computer, and act upon the event or action. Consequently, temporal pattern recognition has attracted more attention in current research area.

Basically, this refers to the recognition of patterns that may or may not be stable over time. Time is what sets temporal pattern recognition aside from the normal blend we introduced previously. Albert Einstein once wrote ‘The only reason for time is to prevent all things from happening at once’. Aside from being a well-known quote it identifies the very reason why temporal pattern recognition is so damn difficult.[2] Speech recognition, control systems (e.g. numerically-controlled machine tools), fault detection, and image processing etc., are all examples of non-stationary systems. Dealing with these types of real-world systems requires special computer processing technique.

2.1.1 Time-Warp Invariant Speech Recognition

Speech recognition has been dawn particular attention in recognition field. One of the most famous temporal benchmark tasks in literature is the Hopfield & Brody speech recognition task where recordings of spoken numbers zero to nine have to be correctly classified [11,12]. The dataset consisted of 10 recordings each of 5 female speakers, totalling to a set of 500 inputs. Each of these recordings was encoded in 40 individual spike trains. Their original solution dubbed ‘mus silicium’ or simply ‘organism’ had the rather unrealistic requirement of a maximum of one spike per input line. The performance of this solution was not surpassed in an Internet competition. It has been chosen as it is a good example of finding patterns in time: high-level features as phonemes and words are to be extracted from the low-level audio tapes. While still being a very difficult task, it does have a certain number of properties that make it simpler than many other dynamic real world applications. Although everybody's pronunciation of words is different, there are identifying properties likely to be present [11, 12]. These properties range from the instants on which frequencies rise and fall to the comparative strength at which they are being produced. Filtering techniques could be used to extract these features, or allowing the recognizer to classify them. However, there is a possibility that not every task has such apparent and not every task contains extractable features that can be identified easily, one of which is the recognition of textures using whiskers [13].

Maass states that even randomly connected circuits of spiking neurons can in theory be used for complicated computational tasks such as time-warp invariant speech
recognition. Maass only said so because such circuits have a natural tendency to integrate received information in such a way that simple linear readouts can be trained to convert the current circuit activity into the desired output for a enormous number of computational tasks. [3]

2.1.2 Analog-Match Problem

Recognizing words independent of a uniform stretch (“uniform time warp”) can in theory be cast as an analog-match problem and transformed into neural variables [4]. It is difficult for a neural system such as a feedback network to classify two temporal sequences $EEEEEFFGGYY$ and $EEFGY$ as “equal” (up to an overall threefold time warp) and yet distinguishing the two sequences from a third sequence, for example $EEEEFFGGGY$. Consequently, time-warp invariant sequence recognition is a major challenge for any recognition systems.

2.2 Hidden Markov Models

Hidden Markov chains was originally introduced and studied in the late 1960s and early 1970s. During the 1980s the models became increasingly popular. Time after time hidden Markov models have been demonstrated to be highly effective in pattern recognition such as speech recognition. The reason for this is two-folded. Firstly, the hidden Markov models are very rich in mathematical structure and hence can form the theoretical foundation for a wide range of applications. Secondly, the models have, when applied appropriately, turned out to be highly successful. Some of the notable applications are speech recognition and bioinformatics in particular protein modelling.

2.2.1 What Is a Hidden Markov Model

Hidden Markov Models (HMM) can be seen as an extension of Markov models to the case where the observation is a probabilistic function of the state, i.e. the resulting model is a doubly embedded stochastic process, which is not necessarily observable, but it can be observed through another set of stochastic processes that produce the sequence of observations. [5]

A hidden Markov model works by building a mathematical structure called a (hidden) Markov chain where each state produces an observation. The observations are noticeable and the goal is to infer the hidden state sequence. For example, the hidden states may represent words or phonemes, and the observations represent the acoustic signal.

Mathematically, a hidden Markov model [6] consists of a set of $N$ nodes, each of which is associated with a set of $M$ possible observations. The parameters of the model include an initial state

$$\pi = [p_1, p_2, p_3, \ldots, p_n]^T$$
with elements $p_n; n \in [1,N]$ which describes the distribution over the initial node set, a transition matrix

$$A = \begin{pmatrix}
a_{11} & a_{12} & \ldots & a_{1n} \\
a_{21} & a_{22} & \ldots & a_{2n} \\
M & M & O & M \\
an_1 & an_2 & \ldots & ann
\end{pmatrix}$$

With elements $a_{ij}$ with $i, j \in [1,N]$ for the transition probability from node $i$ to node $j$ conditional on node $I$, and an observation matrix

$$B = \begin{pmatrix}
b_{11} & b_{12} & \ldots & b_{1n} \\
b_{21} & b_{22} & \ldots & b_{2n} \\
M & M & O & M \\
b_{n1} & bn_2 & \ldots & bnm
\end{pmatrix}$$

with elements $b_{im}$ for the probability of observing symbol $m \in [1,m]$ given that the system is in state $i \in [1,N]$. Rabiner and Juang denote a HMM model parameter set by $\lambda = (A,B,\pi)$ [6]

A hidden Markov model is a good state-based decision method [14, 15] which will make an allowance for internal pre-processing of input data, making it possible that longer-term casual effects and levels of noise are picked up and dealt with within the decision process. In this way, we do not have to introduce a full series of inputs for analysis, but would allow the method to have its own memory. It has been the standard method for speech recognition since the 1970s.

Generally speaking this method can not be applied directly on a raw and noisy input acquired from sensors. It works best on slightly more high-level and de-noised features as frequency diagrams for speech recognition [14]. Once the model has been constructed it will be useful to predict what generator was employed for the current input; in the case of speech recognition this could work on all levels ranging from phoneme detection to full sentence construction. Although hidden Markov models can work very well, for good operation; however, it is necessary to have a reliable set of such high-level and continuous features [15]. As mentioned earlier, in speech recognition tasks these are relatively easy to detect, extract: frequency and peaks are easy to detect and relatively informative to the recognition models. However, this is not suitable for many other temporal pattern problems like the classification of signals from artificial whisker sensors.

There is an enormous volume of literature on the application of hidden Markov models (HMMs) to a wide range of pattern recognition tasks. In general, the suitability and effectiveness of HMMs is unquestionable therefore the models they are recognized as one of the major methods employed by pattern recognition. However, when one looks for papers which focus on fundamental problems such as efficient learning strategies or systematically determining the most suitable structure for a
given problem, the number of significant papers is greatly moderated. So despite the massive usage of HMMs since the models were released, we believe that there is still a great amount of unexplored area.

2.3 Recurrent Neural Networks

Neural networks found in nature are able to process information from various sources concurrently. They evolved in a way to react in real-time environmental changes. To get a better understanding of LSM, the concept of recurrent neuron network is briefly introduced in this section.

Recurrent network (RN) is a model with bi-directional data flow that propagates data from later phase states to earlier states. In contrast, the feed-forward network propagates data linearly from input to output. In computing, feed-forward normally refers to a multi-layer perception network in which the outputs from all neurons go to following but not preceding layers, so there are no feedback loops whereas the RN has got the feedback loop (See Figure 2)

Recurrent networks tend to be more difficult to train than feed-forward networks, as a result of the cycles, though there are still a fair number of algorithms which are frequently used. Some problems are particularly suited for recurrent networks than feed-forward networks, such as time series prediction.

Recurrent neural network plays an important role in neuroscience. Related to the project, there is no clear reason that recurrent neural network could not provide equally useful dynamics at much lower computational rate. [16]

So far, all published experiments using Liquid State Machine for recognition requires the simplest form detection: classification of and between fixed pattern with little noise added to the liquid or input. Classification of pattern generated from templates is much more complex and if necessary, would be easy to apply in real world.
applications.
Chapter 3  Spiking Neural Networks

Over the past hundred years, biological research has built up a huge amount of knowledge about the structure and function of the brain. It is discovered that the elementary processing units in the nervous system are neurons which are connected to each other in some pattern. The biological neurons use short and sudden increases in voltage to send information. These signals are more commonly known as action potentials, spikes or pulses.

Recently, spiking neural networks (SNNs) have been studied by many researchers.[7] This trend is due to quite a lot of biological data indicating that spikes play an important part in the biological information processing, which is in contrast with the view that the average firing frequency is important in biological information processing. Conversely, the effort to use SNN to computation has also increased. Several results show that SNNs are very powerful in computation: It appears that this model has at least the same computational power as neural networks like multi-layer perceptrons and sigmoidal neural networks of a similar size. Furthermore, there are certain functions which require for their computation significantly fewer neurons in SNN than conventional neural network models [7].

A typical neuron can be divided into three functionally distinct parts, called dendrites, soma, and axon; see figure 3.

![Figure 3(A) Schematic drawing of a neuron by Ramon y Cajal. 3 parts of a neuron: dendrites, soma, and axon can be clearly distinguished. The inset demonstrates an example of a neuronal action potential. The action potential is a short voltage pulse of 1 - 2 ms duration and amplitude of about 100 mV. (B). Signal transmission from a pre-synaptic neuron j to a postsynaptic neuron i. The synapse is marked by the dashed circle. The axons at the lower right end lead to other neurons [8]](image)

The encoding type of SNNs is temporal, thus SNN models fits to the study of time-
structured data. (See table 1)

<table>
<thead>
<tr>
<th>Analogue NNs</th>
<th>Recurrent NNs</th>
<th>Spiking NNs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Most common type</td>
<td>• Loops in connections</td>
<td>• Use spikes to communicate</td>
</tr>
<tr>
<td>• Ability to learn</td>
<td>• No layered structure</td>
<td>• More suited for digital hardware</td>
</tr>
<tr>
<td>• Robust to noise</td>
<td>• Temporal processing on the network level</td>
<td>• Temporal processing on the neuron level</td>
</tr>
<tr>
<td>Not good with temporal data</td>
<td>Difficult to train</td>
<td>Difficulty to train</td>
</tr>
</tbody>
</table>

Table 1 A comparison of three different kinds of neural network including analogue, recurrent and spiking models

In addition to this assumption, Hopfield presented a model for radial basis functions (RBFs) with delays [9]. The main idea is that the RBF neurons encode a certain input spike pattern in the delays that connects input neurons to RBF neurons. An input pattern being similar to the encoded pattern, the RBF neuron fires because the delays even out in the delays even out the differences of the firing times of the input neurons. This method is supported from both computational and biological aspects [10].

3.1 Hodgkin-Huxley Model

The Hodgkin and Huxley model (see figure 5) is a classical model that was developed in 1952 as a result of extensive experimental studies on the giant axon of the squid.[8]
A piece of cell membrane is considered to have a capacitance $C$. The ionic currents charge this capacitance:

$$\frac{C \, du}{dt} = - \sum_k I_k + I(t)$$

where $u$ is the voltage across the membrane, $I(t)$ is an external driving current and $\sum I_k$ is the total of the ion currents through the membrane. In this model, there are three types of ion current, indexed by Na, K, and L (leakage):

$$\sum_k I_k = g_{Na} \, m^3 \, h \, (u - V_{Na}) + g_K \, n^4 \, (u - V_K) + g_L \, (u - V_L)$$

The parameters $g_k$ are conductances, $V_k$ are reversal potentials, and $m$, $n$, $h$ are other variables that evolve according to the differential equations:

$$\begin{align*}
\dot{m} &= \alpha_m(u) \, (1 - m) - \beta_m(u) \, m \\
\dot{n} &= \alpha_n(u) \, (1 - n) - \beta_n(u) \, n \\
\dot{h} &= \alpha_h(u) \, (1 - h) - \beta_h(u) \, h
\end{align*}$$

The $\alpha$ and $\beta$ are empirical functions of $u$.

These differential equations practically describe the exact reactions of the neuron’s membrane potential to the arrival of electric stimulation. However, as this practicality costs a large computational power, it makes the model less appropriate for simulations of large networks.

Figure 5 Hodgkin–Huxley fundamental microcircuit. Arrows represent randomly chosen weights of connections.

### 3.2 Integrate-and-Fire Neurons

In this section, we give an overview of integrate-and-fire models. The leaky integrate-and-fire neuron that will be introduced is probably the best-known example of a formal spiking neuron model. In the early publication Maass illustrate the liquid computing theorem with the help of a neural network using Leaky Integrate and Fire Neuron. This model of neuronal spiking provides an analytically tractable formalism of neuronal firing rate in terms of a neuron’s membrane time constant, threshold and refractory period [17].

This model is called “integrate-and-fire” because input signals are integrated summing up and, once a threshold value is reached, the neuron fires, i.e., it emits a spike.[18] The integration (summing up) is leaky since the model resembles the process of charging a capacitor that is at the same time discharged by a resistor.
parallel to the capacitor (the current “leaks” through the resistor).

\[ V_m \]

\[ g_0 \]

\[ V_0 \]

\[ C_m \]

Figure 6 Schematic diagram of the integrate-and-fire model

The circuit is used to model charging of a neuron’s membrane from its resting potential to threshold. \( C_m \) is the membrane capacitance, \( g_0 \) is the membrane resistance.

\[ V(t) \]

\[ I(t) \]

Figure 7 Membrane voltage vs. time. Only the suprathreshold spikes are produced in the output

In the integrate-and-fire model spikes are seen as short current pulses that travel down the axons. Once it arrives at a synapse, the short pulse is transformed by a low-pass filter into a current pulse that charges the next integrate-and-fire circuit. This increase in voltage can be seen as the postsynaptic potential. Once the voltage over the capacitor goes above threshold value, the circuit shunts, sends out a pulse and resets itself.

A lot of research and experiments have shown that the integrate-and-fire model is conceptually and computationally relatively simple, though naturally much heavier than rate coding sigmoidal networks. An advantage of the model is that it is relatively easy to integrate it in hardware, allowing for very fast operation. It has a close relationship with the more general spike-response model (SRM) and can be used like it by rewriting it into the correct kernels that form the core elements of the SRM model [3].

### 3.3 Synapses

In dictionary, synapses are defined as specialized junctions through which cells of the nervous system signal to one another and to non-neuronal cells such as muscles or glands (see figure 8). Synapses form the circuits in which the neurons of the central nervous system are interconnect. They are thus crucial to the biological computations that underlie perception and thought.
They also provide the means through which the nervous system connects to and controls the other systems of the body. Synapses play an important part of communication between nerve cells through axons and dendrites, converting electrical signals into chemical ones.

![Figure 8 different versions of schematic diagram of synapse](image)

Besides signal transducers, real synapses are extremely complicated information preprocessors that show strong dynamics at many different temporal scales [19]. Actually, they act as dynamic memory buffers; changing communication actions depending on the form of the spike train it recently received [20]. Though synapses are mainly employed as static weighing functions in most neural network, research has shown that synapses that model more dynamic features also found in real synapses are computationally much more powerful [22].

### 3.3.1 Synaptic plasticity

Synaptic plasticity is the modification of synaptic weight that occurs during the lifetime of a synapse, often influenced by the activity of the pre- and postsynaptic neurons. Synaptic plasticity in real neurons has been studied for a long time and it has been modelled with many rules.

There are mainly two types of plasticity: short and long term. Short term plasticity only has a very short effect, on the time scale of tens of milliseconds and involves time-dependent operation of the pre-existing resources of the synapse. On the other hand, long term plasticity, lasts from minutes to days, as it requires adjustment of the synaptic resources in a lasting manner.

Synaptic plasticity affects the synaptic weight and efficacy. Short term plasticity is mainly caused by facilitation and depletion of neurotransmitter vesicles: a pre-synaptic AP releases neurotransmitter, but the amount of released depends on the number of available vesicles. When many APs arrive soon after each other, the vesicle pool may be depleted and no transmitter can be set free, as a result, short term depression will occur. Long term synaptic plasticity caused by neural activity results in either long term depression (LTD) or long term potentiation (LTP). Many plasticity rules have been described, but which rules are correct and what the precise mechanisms are is mostly unknown.
Long term plasticity that is based on correlations between pre- and postsynaptic firing is often called Hebbian learning, which is named after Hebb who was the first to recognise this type of plasticity: he suggested that the synaptic efficacy between two neurons is potentiated when the pre-synaptic neuron contributes to the firing of the postsynaptic neuron [19]. Correlated pre- and postsynaptic activity should thus result in LTP, whereas uncorrelated activity should result in LTD. Although classical Hebbian learning rules using firing rates have proven to be useful, research in the last ten years has indicated that 18 precise spike timing is often very important and the term spike timing dependent plasticity has come into existence.
Chapter 4  Liquid State Machines

Maass et al [23] have proposed a model called the Liquid State Machine which allows for real-time computing by employing continuous perturbations in a heterogeneous dynamical system. The basic idea of this model is to use a high dimensional dynamical system and have the inputs continuously perturb it. If the dynamics are sufficiently complex, the LSM should act as a set of filters projecting the inputs into a higher dimensional space.

The LSM is a novel approach towards computation: it uses the internal dynamics of a recurrent spiking neural network to carry out computations on its input. The internal state of the SNN (called the liquid) serves as input for the so-called readout function. The liquid itself does not generate any output; it merely serves as a 'reservoir' for the inputs. The readout then looks at the liquid state (the response of the liquid to a certain input), and computes the output of the LSM.

An analogy (and also an explanation for the name Liquid State Machine) can be found in a real liquid: it also consists of a collection of elements that exert an influence on their immediate neighbours. And, like in the case of the LSM, external stimuli or 'inputs' - e.g. a stone that is thrown into a pond - remain detectable for a long time after they have started - like ripples on the same pond. The analogy goes only so far of course. Contrary to the LSM, water molecules do not compute anything. Nonetheless, a Liquid State Machine using the real liquid (a box of water), has been successfully built and tested on a simple speech recognition problem.

The inputs and outputs of LSMs are arrays of time series. Comparing to Turing computation, this model facilitates the analysis of continuous streams of input.[23] Given a time series of input, the machine can produce a time series of behaviors as output. To get the desired behaviors, one will have to adjust the weights on the links between the core and the output. However, this machine does not require a task-dependent construction of a neural circuit; in other words, the liquid states and the transitions between them need not be customized for a specific task.

4.1 Formal Definition

A Liquid State Machine comprises three parts, an input layer, a large randomly interconnected unit which has the intermediate states transformed from input, and an output layer. As the name of the model hints, they use the microcircuit as a “liquid filter” that serves as an unbiased analogy fading memory about current and preceding inputs to the circuit. Typically recurrent neural nets that employ Leaky Integrate and Fire Neurons (Leaky I&F) are used in these machines. The basic structure is depicted in Figure 9.
The liquid filter unit $L$ serves like excitable medium core to pre-process the input $u(.)$ and transforms the input into liquid states $x(t)$:

$$x(t) = (L u)(t)$$

Then the temporal features extracted are passed to the readout unit that is said to be a “memory-less function” [1] $f$ maps the liquid state $x(t)$ at time $t$ into the output

$$v(t) = f(x(t)).$$

“It is simply assumed that the liquid integrates input $u(t')$ for $t' < t$ into liquid states $x(t)$ for all times $t$.”

![Figure 9 Schematic principle of the liquid state machine. A continuous stream of liquid](image)

### 4.1.1 Existing Similar Approach

An approach very similar to the LSM approach has been independently explored by Jaeger in the context of artificial neural networks [24]. It is called Echo State Net (ESN) which “allow for universal real-time computation without stable state or attractors on continuous input streams” [24]. This model seems nearly identical as the LSM but just from an engineering point of view. Both of them are using the dynamics of recurrent neural networks for pre-processing input and train extra mechanisms for obtaining information from the dynamic state of these networks.

However, there are slight distinctions between them:

1. **LSM focuses on modelling dynamical and representational fact in biological neural network, whereas ESN lies in the area of engineering application**

2. **The liquid network in LSMs is typically made fro, biologically more**
adequate, spiking neuron models, whereas ESN reservoirs are typically made up from simple sigmoid units[24]

c) LSM research considers a variety of readout mechanisms, including trained feed-forward networks, whereas ESM typically make do with a single layer of readout units[25]

Jaeger[24] also found that he could significantly increase the performance of an Echo State Machine by using recurrent networks of continuous leaky-integrator neurons instead of standard recurrent sigmoid networks. On the other hand, there is not much evidence showing that LSM also performs better by using Leaky Integrate and Fire Neuron.

4.2 Computational Power

By analysing the computational power of LSMs, we can eventually characterize which mappings from input time series to output time series can be computed by this model. Maass proposed that LSMs have universal computational power for computation on time series[23]. This hypothesis assumes that LSMs have two abstract properties: separation property (SP) and approximation property (AP)

4.2.1 Separation and Approximation Property

The most important feature of the liquid is to react in a quite different way to differentiate input sequences. The amount of distance generated between those is named the separation property (SP) of the liquid.

The SP (see fig. 9) reflects that the liquid has the ability to create different trajectories of internal states for each of the input classes. The ability of the readout units to distinguish these trajectories, and connect them to the target outputs is called the approximation property (AP). This property depends on the adaptability of the chosen readout units, whereas the SP is based directly on the liquid’s complexity. As we mentioned above, Jaeger[24] found that he could significantly increase the performance of an Echo State Machine by using
recurrent networks of continuous leaky-integrator neurons instead of standard recurrent sigmoid networks. Maass [26] found very similar results when he compared the performance of LSMs using liquids of spiking neural networks with either static weighing connections or the more realistically dynamic synapses. We will not be surprised to see that more dynamic connections increases the computational power is not surprising if we understand the fact that using these neurons and synapses hugely increases the temporal dynamics of the medium, making input trajectories more likely to be unique and consequently make recognition of different input streams easier. Regarding the separation property, it can be predicted that there seems to be a strong link between the computational power of the full system and the non-linearity of the liquid networks [27]. This could also be the main reason why separation capability of neural network has been used as a standard test for their computational capabilities
Chapter 5  Learning Algorithms

5.1  Parallel Delta Rule

Recently, Auer [24] introduced the p-delta rule for parallel perceptrons, neural networks of a single layer of perceptrons. Its performance is similar to that of other state-of-the-art algorithms, but is much easier to build: only one layer of perceptrons and just one global 2-bit communication signal is required.

The delta rule is a rule for updating the weights of the neurons in a single-layer perceptron. For a neuron $j$ with activation function $g(x)$ the delta rule for $j$’s $i$th weight $w_{ji}$ is given by

$$
\Delta w_{ji} = \alpha (t_j - x_j)g'(h_j)x_i
$$

where $\alpha$ is a small constant, $g(x)$ is the neuron's activation function, $t_j$ is the target output, $x_j$ is the actual output, and $x_i$ is the $i$th input. The delta rule is commonly stated in simplified form for a perceptron with a linear activation function as

$$
\Delta w_{ji} = \alpha (t_j - x_j)x_i
$$
5.1.1 Derivation of Delta Rule

The delta rule is derived by attempting to minimize the error in the output of the perceptron through gradient descent. The error for a perceptron with \( j \) outputs can be measured as

\[
E = \sum_j \frac{1}{2} (t_j - x_j)^2
\]

In this case, we wish to move through "weight space" of the neuron (the space of all possible values of all of the neuron's weights) in proportion to the gradient of the error function with respect to each weight. In order to do that, we calculate the partial derivative of the error with respect to each weight. For the \( i \)th weight, this derivative can be written as

\[
\frac{\delta E}{\delta w_{ji}}.
\]

Because we are only concerning ourselves with the \( j \)th neuron, we can substitute the error formula above while omitting the summation:

\[
\frac{\delta E}{\delta w_{ji}} = \frac{\delta \left( \frac{1}{2} (t_j - x_j)^2 \right)}{\delta w_{ji}}
\]

Next we use the chain rule to split this into two derivatives:

\[
= \frac{\delta \left( \frac{1}{2} (t_j - x_j)^2 \right)}{\delta x_j} \frac{\delta x_j}{\delta w_{ji}}
\]

To find the left derivative, we simply apply the general power rule:

\[
= - (t_j - x_j) \frac{\delta x_j}{\delta w_{ji}}
\]

To find the right derivative, we again apply the chain rule, this time differentiating with respect to the total input to \( j \), \( h_j \):

\[
= - (t_j - x_j) \frac{\delta x_j}{\delta h_j} \frac{\delta h_j}{\delta w_{ij}}
\]

Note that the output of the neuron \( x_j \) is just the neuron's activation function \( g() \) applied to the neuron's input \( h_j \). We can therefore write the derivative of \( x_j \) with respect to \( h_j \) simply as \( g() \)'s first derivative:

\[
= - (t_j - x_j) g'(h_j) \frac{\delta h_j}{\delta w_{ij}}
\]

Next we rewrite \( h_j \) in the last term as the sum over all \( k \) weights of each weight \( w_{jk} \) times its corresponding input \( x_k \):

\[
= - (t_j - x_j) g'(h_j) \frac{\delta (\sum_k x_k w_{jk})}{\delta w_{ij}}
\]

Because we are only concerned with the \( i \)th weight, the only term of the summation that is relevant is \( x_i w_{ij} \). Clearly,
\[ \frac{\delta x_i w_{ji}}{\delta w_{ji}} = x_i, \]
giving us our final equation for the gradient:
\[ \frac{\delta E}{\delta w_{ji}} = -(t_j - x_j) g'(h_j)x_i. \]

As noted above, gradient descent tells us that our change for each weight should be proportional to the gradient. Choosing a proportionality constant \( \alpha \) and eliminating the minus sign to enable us to move the weight in the negative direction of the gradient to minimize error, we arrive at our target equation:
\[ \Delta w_{ji} = \alpha (t_j - x_j) g'(h_j)x_i. \]

5.2 Backpropagation

Backpropagation is a supervised learning technique used for training neural networks. It is most useful for feed-forward networks (networks that have no feedback, or simply, that have no connections that loop). The term is an abbreviation for "backwards propagation of errors". Backpropagation requires that the transfer function used by the artificial neurons (or "nodes") be differentiable.

The gist of the technique is as follows:

1. Present a training sample to the neural network.
2. Compare the network's output to the desired output from that sample. Calculate the error in each output neuron.
3. For each neuron, calculate what the output should have been, and a scaling factor, how much lower or higher the output must be adjusted to match the desired output. This is the local error.
4. Adjust the weights of each neuron to lower the local error.
5. Assign "blame" for the local error to neurons at the previous level, giving greater responsibility to neurons connected by stronger weights.
6. Repeat the steps above on the neurons at the previous level, using each one's "blame" as their error.

As the algorithm's name implies, the errors (and therefore the learning) propagate backwards from the output nodes to the inner nodes. So technically speaking, backpropagation is used to calculate the gradient of the error of the network with respect to the network's modifiable weights. This gradient is almost always then used in a simple stochastic gradient descent algorithm to find weights that minimize the error. Often the term "backpropagation" is used in a more general sense, to refer to the entire procedure encompassing both the calculation of the gradient and its use in stochastic gradient descent. Backpropagation usually allows quick convergence on satisfactory local minima for error in the kind of networks to which it is suited.
5.2.1 Levenberg-Marquardt (trainlm)

As in the experiment environment CSIM, backpropagation algorithm is implemented by “trainlm” function, this section will give a brief introduction to this second-order nonlinear optimization technique. The LM algorithm is widely applied to many different domains and is faster and produces better results than other training methods. To update weights and biases, the LM algorithm uses an approximation to the Hessian matrix. It was designed to approach second-order training speed without having to compute the Hessian matrix. When the performance function has the form of a sum of squares (as is typical in training feed-forward networks), then the Hessian matrix can be approximated as

\[ \mathbf{H} = \mathbf{J}^T \mathbf{J} \]

and the gradient can be computed as

\[ \mathbf{g} = \mathbf{J}^T \mathbf{e} \]

where \( \mathbf{J} \) is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and \( \mathbf{e} \) is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

\[ x_{k+1} = x_k - [\mathbf{J}^T \mathbf{J} + \mu \mathbf{I}]^{-1} \mathbf{J}^T \mathbf{e} \]

When the scalar \( \mu \) is zero, this is just Newton's method, using the approximate Hessian matrix. When \( \mu \) is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift towards Newton's method as quickly as possible. Thus, \( \mu \) is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function will always be reduced at each iteration of the algorithm.

This algorithm appears to be the fastest method for training moderate-sized feed-forward neural networks (up to several hundred weights). Since the solution of the matrix equation is a built-in function in Matlab, its implementation in Matlab is quite efficient.

However, Levenberg-Marquardt algorithm’s major drawback is that it requires the storage of some matrices that can be quite large for certain problems. The size of the Jacobian matrix is \( Q \times n \), where \( Q \) is the number of training sets and \( n \) is the number of weights and biases in the network. It turns out that this matrix does not have to be computed and stored as a whole. For example, if we were to divide the Jacobian into
two equal submatrices we could compute the approximate Hessian matrix as follows:

\[
H = J^T J = \begin{bmatrix} J_1^T & J_2^T \end{bmatrix} \begin{bmatrix} J_1 \\ J_2 \end{bmatrix} = J_1^T J_1 + J_2^T J_2
\]

Therefore, the full Jacobian does not have to exist at one time. The approximate Hessian can be computed by summing a series of sub-terms. Once one sub-term has been computed, the corresponding sub-matrix of the Jacobian can be cleared. Even if you use memory reduction, the Levenberg-Marquardt algorithm will always compute the approximate Hessian matrix, which has dimensions \( n \times n \). If your network is very large, then you may run out of memory.

### 5.3 Hebbian learning

As we mentioned above, Hebbian learning is a learning rule proposed in 1949 by Donald Hebb. It is a hypothesis for how neuronal connections are enforced in mammalian brains; actually it is also a technique for weight selection in artificial neural networks.

From the point of view of artificial neurons and artificial neural networks, Hebb's principle can be described as a method of determining how to alter the weights between model neurons. The weight between two neurons will increase if the two neurons activate simultaneously; it is reduced if they activate separately. Nodes which tend to be either both positive or both negative at the same time will have strong positive weights while those which tend to be opposite will have strong negative weights. It is sometimes stated more simply as "neurons that fire together, wire together."

This original principle is perhaps the simplest form of weight selection. While this means it can be relatively easily coded into a computer program and used to update the weights for a network, it also prohibits the number of applications of Hebbian learning. Hebbian learning involves weights between learning nodes being adjusted so that each weight better represents the relationship between the nodes. As such, many learning methods can be considered to be somewhat Hebbian in nature.
Chapter 6 Noise

Noise sensitivity is an important criterion to measure how well a system is performing the tasks in the real world. Aside from the speech-recognition task where inputs were jittered and performance improved [23], no results have been published yet on what effect noise has on general LSM performance. Even so, the results acquired by those who took 'liquid' in LSM literally indicate a high noise resistance [28]. Their setup consisted of a glass bucket of water on top of an overhead projector and a camera recording the obstruction patterns created by disturbing the water’s surface with motors. The record was then fed into a perceptron for classification. Despite the very noisy environment they were able to solve not only XOR problem, but could also distinguish ones and zeros from the Hopfield-Brody [11,12] task. Research would have indicated that Liquid State Machines can perform much better that we thought. However, the experiment results particularly focusing on noise resistance have not been published.
Chapter 7 Generating Liquid State Machines

In order to examine the effectiveness of Liquid State Machines for real-life recognition tasks, a set of experiments have been set to illuminate some certain properties of this model. The information about how to build a LSM is introduced in this chapter.

7.1 Environment

7.1.1 CSIM

The CSIM is a simulator for modelling a large heterogeneous network which consists of a number of different model neurons and synapses. The size of this network is up to 10,000 neurons along with a few millions of synapses. However, the actual size of this network depends on the RAM on running machines. This simulation is developed by C++ with an MEX interface to Matlab.

7.1.2 Circuit-Tool

The Circuit-Tools contains a set of Matlab objects and scripts. These objects and scripts allow the multi-‘column’ neural microcircuits construction to be existed with distributed deployed parameters. The neural microcircuit models are simulated by using CSIM.

7.1.3 Learning-Tool

Learning-Tool is a tool for analyzing the neural microcircuit models of capability of real-time computing capability. It contains a set of Matlab scripts, and is based on a new theoretical framework which is known as the Liquid State Machine.
7.2 Other Issues

7.2.1 Simulation

Though spiking neurons can comparatively easily be implemented in hardware, simulation of this kind of neurons is a big challenge as it requires much more computational power than traditional artificial neurons do [30].
7.2.2 Input and Output

A proper transformation of input and output make a recognition process much easier and faster. However, before input processing, the data usually will have to be translated into some signals that the system can understand. In a similar way, output of the process will have to be decoded before they could be fully understood by us.
Chapter 8  Experimental Design

8.1 Overview

This section of the dissertation is concerned with identifying what information is required to evaluate how well this model performs. A number of the algorithms have been mentioned in the previous chapters. This section discusses how these algorithms will be used to generate a Liquid State Machine in CSIM.

8.2 Experiment Objectives

In order to enable well-conducted experiments, a number of basic objectives were proposed. They were identified as follows:

1. To ensure that all experiments carried out are fair and adequate, statistical techniques will be employed
2. To show a fair comparison between the different algorithms, the same input (randomly generated) will be applied
3. In order to give a reasonable evaluation, a similar of liquid structure (typically, 3*3*15 spiking neurons) that was used by Maass will be used
4. To that the user of the system should be able to run the tests with the minimal amount of effort and should be able to control most aspects of the simulation.
5. To gain clearer representations of data and results, graphical representations will be preferred.

8.3 Experiment Key

This section explains the key performance functions that have been used in the experiments. These functions are the key criteria that measures the performance

- MAE: mean absolute error
- Score: error score, it takes into account false and positive negatives
- CC: correlation coefficient
- MSE: a network performance function. It measures the network's performance according to the mean of squared errors

The mean absolute error is the positive difference between the activation values of target and output sequences of the validation set divided by the number of neurons in the input/output layer and the length of the sequence.
A problem which arises if only the mean absolute error is used for evaluation is that also networks with nearly no output activation produce a low mean absolute error. The correlation coefficient measures the linear dependency of two variables. If the value is zero two variables are not correlated. The correlation coefficient is calculated in similar way as the mean absolute error. Therefore the higher the coefficient the higher the probability of getting a correlation as large as the observed value without coincidence involved. A relation between mean absolute error and correlation coefficient could exist. A high correlation coefficient may indicate a low mean absolute error.

8.4 Experiment Scenario

In this section we will explain the experiment by means of scenarios. The experiments train a readout neuron modelled as a threshold gate to classify a spike train. This readout unit will get its input from a neural microcircuit modelled as a pool of leaky-integrate-and-fire neurons which is stimulated by the input spike train (which should be classified). The setup is shown in figure 12

Precisely, two Poisson spike trains (frequency 20 Hz, length 0.5 sec) are first generated, and fixed as templates 0 and 1. The actual input spike train is generated as jittered versions of a template by varying each spike by a random drawn amount (Gaussian distribution with zero mean and a given STD; this STD is called jitter (default jitter=4ms)). Threshold gate is designed to output the number (0 or 1) of the (random chosen) template from which the input spike train was generated.

The main process which will be used to train the readout is described below:

1. Identify the neural microcircuit to be analyzed
2. Record spike responses of the neural microcircuit caused by different training inputs drawn from an appropriate input distribution.
3. Translate the spike responses into states $x(t_k)$ at various sample time points $t_k$ by some low-pass filtering to get a somewhat signal. This models the effect of
spike transmission through a synapse to its post-synaptic neuron.

4. Apply a supervised learning algorithm to a set of training examples such that the actual outputs are as close as possible to the target values given by the target function.

5. Evaluate the performance of the trained readout (i.e. the threshold gate) on an independent set of test inputs (which are usually drawn from the same distribution as the training inputs).

The above description of the basic process clearly introduced all the basic concepts we need to know to understand how the experiments will be carried out:

*Input Distribution*: The distribution from which the training (and test) inputs are drawn. In our example the input distribution is defined by the following simple procedure (for fixed templates 0 and 1):

1. Randomly choose template 0 or 1
2. Add noise (jitter) to each spike in the template

*Neural Microcircuit*: The circuit which receives the input and whose response is recorded and analysed (in our example this is a network of 135 leaky-integrate-and-fire neurons). Response of the Microcircuit: The response (output) of the neural microcircuit (in our example the 135 spike trains produced by the microcircuit model). State of the Microcircuit: The transformed (smoothed) response (output) of the neural microcircuit (in this example this corresponds to a low-pass filtered (30ms) version of the spike trains).

*Sample Time Points*: Since we can only handle finite sets of training examples we must define time points at which we want to sample the state of the microcircuit (in this example we will sample the states every 25 ms).

*Readout Function*: A parameterized function/device which gets as input the circuit states (or in some cases directly the circuit response) and computes the outputs of the system (in this example a threshold gate).

*Target Function/Filter*: A function which defines for each input time series the target output time series of a readout function. In mathematical terms this should be a target filter since we are talking about computations on time series.

*Supervised Learning Algorithm*: By means of such algorithm the parameters of the readout— and only the readout — are adjusted such that the actual output of the readout matches as close as possible the target output.

*Training Set*: Set of inputs used to determine the parameter of the readout.

*Test Set*: Set of inputs different to the training set which is used to asses the performance of the trained readout.
Chapter 9  Results and Analysis

9.1 Experiment 1: Comparison of Learning Algorithms

9.1.1 Method

By processing the same input, we compared the MAE, MSE, CC and SCORE of different learning rules to get a general conclusion of how well they train the network. The following screen shots show the whole process of one single run. The first one shows the process of generating the neuron pools and connecting them together. Meanwhile, the second one mainly displays the training process and results.
9.1.2 Results and Analysis

The summary of the results are shown in the following table.

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
<th>Linear Classification</th>
<th>P-delta rule</th>
<th>Linear Regression</th>
<th>Backpropagation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Training</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td>0.5364</td>
<td>0.7431</td>
<td>0.5605</td>
<td></td>
<td>0.8764</td>
</tr>
<tr>
<td>MAE</td>
<td>0.2423</td>
<td>0.1624</td>
<td>0.3745</td>
<td></td>
<td>0.1375</td>
</tr>
<tr>
<td>MSE</td>
<td>0.2423</td>
<td></td>
<td>0.1713</td>
<td></td>
<td>0.0594</td>
</tr>
<tr>
<td>SCORE</td>
<td>0.7633</td>
<td>0.3431</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td>0.5255</td>
<td>0.7273</td>
<td>0.5429</td>
<td></td>
<td>0.8405</td>
</tr>
<tr>
<td>MAE</td>
<td>0.2326</td>
<td>0.1627</td>
<td>0.3782</td>
<td></td>
<td>0.1533</td>
</tr>
<tr>
<td>MSE</td>
<td>0.2326</td>
<td></td>
<td>0.1742</td>
<td></td>
<td>0.0751</td>
</tr>
<tr>
<td>SCORE</td>
<td>0.8515</td>
<td>0.3654</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2 Summary of performance functions that measure different learning rules

As we can see in the table, the backpropagation performs much better than the other algorithms by having a 0.0594 training MSE and a 0.0751 testing MSE. (Figure 14 below shows the MSE versus training epoch plot – both the training data MSE and validation data MSE curves are shown. The MSE reached 0.01811 and 0.1522 before a validation stop occurred.)
However, one big problem arose in the experiment is that the training time it costs is exceptional longer than the others. That is the main disadvantage of this algorithm.

In general, parallel delta rule performs quite well in the experiment. Although, it does not perform as well as backpropagation, its performance is comparable to that of other two algorithms, also, it is much easier to build: only one layer of perceptrons and just one global 2-bit communication signal is required.

As expected, linear classification performs the worst among these four. It has a much larger MSE than the others. Although it has the highest SCORE, it does have its advantages: simple and fast.

Compared to p-delta rule, linear regression has a better performance while it has a SCORE 0. In practice, it is easier than backpropagation to be built. Consequently, linear regression will be a better choice to be the main learning algorithm.

However, the performance of the learning algorithms partially depends on the input the network receives. (See figure 15) As we can see below, backpropagation still has the highest performance which mean it has the lowest MAE and MSE. However, linear classification do not perform as bad as it should be. It performs significantly better than the other two rules.
Figure 14: A random readout figure shows that linear classification has a higher performance than p-delta and linear regression.
9.2 Experiment 2: Noise Sensitivity

9.2.1 Method

By varying the amount of noise that added to the data before training, the performance of a LSM can be analyzed.

9.2.2 Results

<table>
<thead>
<tr>
<th>Noise</th>
<th>0.01</th>
<th>0.05</th>
<th>0.1</th>
<th>0.2</th>
<th>0.5</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td>CC</td>
<td>0.6134</td>
<td>0.4537</td>
<td>0.7222</td>
<td>0.7222</td>
<td>0.8683</td>
<td>0.5037</td>
</tr>
<tr>
<td></td>
<td>MAE</td>
<td>0.2579</td>
<td>0.3232</td>
<td>0.1812</td>
<td>0.1812</td>
<td>0.0997</td>
<td>0.3206</td>
</tr>
<tr>
<td></td>
<td>MSE</td>
<td>0.1653</td>
<td>0.2199</td>
<td>0.1254</td>
<td>0.1254</td>
<td>0.0636</td>
<td>0.1968</td>
</tr>
<tr>
<td></td>
<td>SCORE</td>
<td>0.5724</td>
<td>1.0726</td>
<td>0.3844</td>
<td>0.3844</td>
<td>0.1805</td>
<td>0.7858</td>
</tr>
<tr>
<td>Testing</td>
<td>CC</td>
<td>0.5589</td>
<td>0.3465</td>
<td>0.6424</td>
<td>0.6424</td>
<td>0.8583</td>
<td>0.5398</td>
</tr>
<tr>
<td></td>
<td>MAE</td>
<td>0.2552</td>
<td>0.3457</td>
<td>0.2066</td>
<td>0.2066</td>
<td>0.0841</td>
<td>0.2731</td>
</tr>
<tr>
<td></td>
<td>MSE</td>
<td>0.1911</td>
<td>0.2679</td>
<td>0.1633</td>
<td>0.1633</td>
<td>0.0746</td>
<td>0.2282</td>
</tr>
<tr>
<td></td>
<td>SCORE</td>
<td>0.6686</td>
<td>1.8178</td>
<td>0.5114</td>
<td>0.5114</td>
<td>0.1741</td>
<td>0.9496</td>
</tr>
</tbody>
</table>

Table 3 Summary of performance functions that measure the performance over noises.

As we can see in the table above, the liquid state machine does not change its performance in any pattern with the noise varying. The best SCORE is 0.5114 when the noise value is 0.2 while the worst one is 1.8178 while the noise is reasonably small at 0.05. It can draw a conclusion that LSM can perform reasonably well in noisy situations. This is an important property of LSM which can make it exceptionally useful in recognition area in future; because how to handle noisy situations is still a big problem faced by recognition systems. For example, most speech recognition systems nowadays still can not work appropriately as if one coughs when he is making the speech.
Chapter 10 Conclusion and Future Application

We set up a few experiments to investigate the possibilities of using a Liquid State Machine with networks of randomly connected integrate-and-fire spiking neurons as ‘liquid’ pre-processor for temporal pattern recognition. Our first set of experiments further confirmed that linear regression is the most appropriate learning algorithm. The main achievement of this project is that it has been possible show that liquid state machine can perform noisy recognition tasks reasonable well. It is theoretically promised that Liquid State concept can performs well in a 'real-world' context. And The LSM is well suited for recognition of various problems.

More experiments could be done to show if biologically realistic configuration gives the best recognition results. Possibly, experiments could be done to show if the LSM can predict non-linear movement. If it could be proved to be realistic, this model could be applied to robots.
Reference


[16] H. Jaeger. The "echo state" approach to Analyzing and Training Recurrent Neural Networks

[18] Doron Tal and Eric L. Schwarts. *Computing with the leaky integrate and fire neuron: logarithmic computation and multiplication*


Chapter 11 Webography

    http://en.wikipedia.org/wiki/Main_Page
    http://www.igi.tugraz.at/maass/
    http://www.lsm.tugraz.at/index.html