Utrecht University
Faculty of Humanities
Department of Philosophy

Master of Science Thesis

Structured Liquids in Liquid State Machines

by

Jonas Matser

Supervisors: Jan Broersen, Gerard Vreeswijk and Clemens Grabmayer

Utrecht, 2010

Abstract

In this thesis we look at improving the performance of Liquid State Machines on a speech recognition and a music recognition task, by making structural changes to the liquid. The changes are designed, rather than evolved or learnt, to have more insight into what the effect of particular changes on performance are. We conclude that the effect of any designed structural changes is far outweighed by the random generation of liquids and that the evaluation of LSMs should focus on using sufficiently complex and preferably realistic tasks.

Keywords: Liquid State Machines, Speech Recognition, Music Recognition, Spiking Neural Networks

Contents

Chapter 1

Introduction

1.1 Liquid state machines

Imagine a bucket of water, into which we throw, one-by-one or a few at a time, a handful of pebbles. As the pebbles hit the water, they cause ripples to appear on the surface. The ripples from different pebbles create varying patterns of interference, but all of them fade out after a while. From the pattern of ripples on the surface at any point in time, we are able to make some inferences about the timing and locations of the pebbles dropped into the water.

The above analogy illustrates the basic idea of reservoir computing. We use a reservoir as a fading memory to process the input. Then we look at the state of the reservoir to draw conclusions about the input.

An important choice in reservoir computing is the type of reservoir we use, as an actual simulation of water dynamics may not be the best choice. Liquid state machines (LSMs) (introduced by Maass, Natschläger and Markram in 2000 [1]) make use of a spiking neural network (SNN) as a reservoir. Maass et al. call this reservoir the liquid. A reservoir computing algorithm with a different reservoir can be found in Jaeger's 2001 paper [2], called an echo-state network.

A spiking neural network consists of a number of neurons, each with their own activation state called the membrane potential. This is a reference to biology, where the activation is actually a potential between the cytoplasm or intracellular fluid (the fluid inside the cell) and the extracellular fluid (the fluid outside the cell), which are separated by the cell membrane.

There are connections from one neuron to another at points called synapses. Any

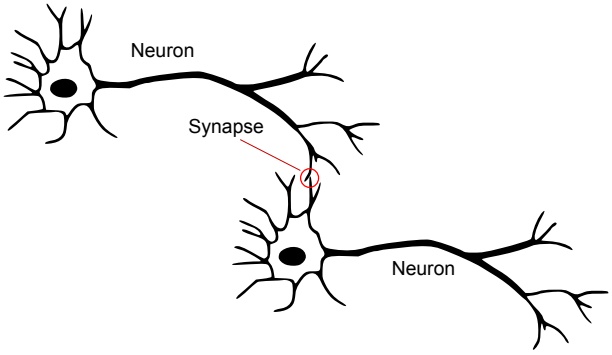


Figure 1.1: Synapse

input given to a neuron in the network (be it external, or from another neuron) changes its membrane potential. If this makes the neuron's membrane potential cross a certain value called the 'firing threshold', the neuron fires, sending a signal to any neurons to which it has connections. The neurons that receive this signal are called the post-synaptic neurons for this cell. This is how signals propagate through the network, depending on how the neurons are connected.

The membrane potential has a resting state, called the resting potential, to which it always slowly returns if the neuron doesn't receive any input. Because this causes old information to slowly make room for new information, this is called a 'fading memory'.

The real novelty of reservoir computing is not in the above but in how we use the state of the reservoir to draw conclusions about the input. As we will see later, training spiking neural networks to perform specific tasks is a complex and time-consuming process. Instead, we use a simpler technique from statistical learning to perform our task. This technique is applied to the state of the reservoir, instead of the input itself. This step is called the readout.

For the readout we take snapshots of the state of the reservoir at certain points during the processing of a sample. These snapshots are called readout moments. Then we use a readout function, usually a comparatively simple feed-forward neural network (FNN), to draw conclusions about the sample based on the readout moment. For a classification task, this means classifying the sample as belonging to a certain category. To do this correctly, the FNN needs to be trained, so we use a collection of training samples with known classifications to improve the performance of the readout.

So for a LSM, we only need to train the comparatively simple FNN of the readout function and not the complex spiking neural network we use as a reservoir.

However, we still get the computational power and flexibility of input of a SNN. Because training algorithms for SNNs are both complex and take a lot of time to run, this alone already makes LSMs an attractive option, because it can save a lot of time.

One of the benefits of this combination of spiking neural networks and feed forward neural networks, is that we can effectively deal with a specific class of problems, that FNNs on their own have trouble dealing with. That class of problems is time series. A time series problem, in this context, means a problem that requires the combination and integration of information over a span of time to solve.

Two examples of time series problems are speech and music recognition. In both of these cases, it isn't enough to look at a single slice of time to know what words are being spoken or what music is being played. Instead, a larger part of the sample has to be examined to be able to come to any conclusions. Liquid state machines work well for these kinds of problems, because of the fading memory: information from earlier time steps still exists in the liquid and is combined with information entering the liquid at a later time.

It might also be possible to use only a FNN for these tasks, for example by flattening the time series data to a single sample containing data for a number of fixed time points. However, using this trick forces these same time points onto every sample, regardless of length. This necessitates choosing a maximum length for the samples in advance. LSMs can handle any sample length and are even able to give intermediate results while only a part of the sample has been seen.

1.1.1 Liquid structure

In their original paper [1], Maass et al. used liquids of 135 neurons, shaped like a column of 3 by 3 by 15 neurons. The spatial location of the neurons determines the chance of a connection between a pair of neurons: neurons that are close together have a higher chance of being connected than neurons that are further apart. In this way there should still be some structure in the propagation of information through the liquid, while the recurrent connections keep the signal from simply stopping at the end of the column.

These randomized connections work fine for all kinds of tasks and are actually one of the strong points of LSMs: the liquid part doesn't need to be task specific. Indeed, because it isn't task specific, it's possible to use the same liquid to perform multiple different tasks in parallel (more on this later).

While randomized connections may perform reasonably well for any and all tasks,

there is a strong intuition that task-specific liquids should be able to improve LSM performance. Just like it's possible to train a spiking neural network to perform a certain task, it might be possible to train the SNN that makes up a liquid to improve the readout performance in a certain task. As we will show later, picking the best liquid from a large number of randomly generated ones already increases performance. In a sense, training the SNN is simply a more cost-effective way of finding better liquids. However, the creation of these task specific liquids shouldn't be too time consuming, because this would lessen one of the big advantages of LSMs.

There has been some research in this direction, of which I will specifically mention the work done by another student under Marco Wiering's supervision, Stefan Kok. In his thesis [3], Kok describes how he applied both Reinforcement Learning techniques and Evolutionary Algorithms on the liquid's structure in order to improve LSM performance.

Both reinforcement learning and evolutionary algorithms require one to evaluate the fitness of each of the liquids created. They also require a large number of iterations in order to be effective. Unfortunately fully training and testing an LSM takes quite some time, so Kok used a different approach.

Instead of training and testing the LSM, he evaluates the so-called separation property of the liquid. The separation property is one of the two fundamental properties of LSMs introduced by Maass et al. Simply put: it's the ability of the liquid to show different activity for different samples, thereby separating them. Maass' separation measure is the euclidian distance between the same readout moments for different samples. Not having to train and test the LSM to evaluate it, but just calculating Maass' separation property, saves a lot of time, which allowed him to spend more computation time on reinforcement learning and evolutionary algorithms.

The other fundamental property of LSMs is approximation. Approximation is the readout's ability to map the liquid activity to the desired outcomes. Because separation and approximation taken together are the extent of the work performed by LSMs it was expected that improving either of these properties would improve the performance of the LSM as a whole.

Unfortunately, Kok's research didn't give the expected improvements in LSM performance. There are a few different possible explanations:

- The proposed measure for LSM performance isn't good enough. Alternative measures can be found in Nortons 2008 thesis [4].
- The readout function doesn't take advantage of the improved separation by

the liquid. This could be due to the nature of feed-forward neural networks, that are often used. One way to test this, would be to try other types of readout functions;

- The fact that readout moments are only constructed periodically from the liquid activity and that they only contain the spikes at that point in time, means a lot of information is discarded. It does seem unlikely that this caused Kok's results, because he measured the separation improvement by looking purely at the readout moments, at a stage where timing doesn't play any part;

- Separation as it was calculated is very sensitive to stretching or shifting of the input signal, because it compares readout moments in pairs. The measure could be improved by using minimum edit distance or other measures less sensitive to these kinds of distortion. Alternative measures for separation can be found by looking at measures for spike train distance, for example the acticle by J.D. Victor at al [5].

As a different approach to improving LSM performance, we will look at two designed structural changes to the liquid, as opposed to evolved or trained.

We have considered measuring the performance of our changes using the separation property as well, but decided that getting an accurate performance measurement was more important to us. Using the separation property as a performance measure makes that hard, as its exact influence isn't clear. This does come at the cost of fairly long running times for our experiments. The approximation property doesn't play a role in our research as we are only looking at improving the liquid and not the readout.

1.2 Research Question

Although the random creation of the liquid is central to the ease of training LSMs, it is also clear that this will usually not give the best possible performance on a task. Having seen the limited success of evolved and learned liquid structure, we will look at structures designed by humans. Our research question then becomes:

Can we improve the classification performance of Liquid State Machines on a selection of tasks by employing the following two structural variations of the liquid part of the LSM?

1. The introduction of spiking neurons working on different time-scales. Normally, the neurons in the LSMs liquid are all the same. The only difference between two neurons is the connections each has to other neurons in the liquid. We will look at a structural variation consisting of a new class of neurons called 'slow neurons'.

As the name suggests, these neurons will be slower than the standard neurons. This means that they take longer before they fire and also take longer to return to their resting state. By slowing these neurons, they collect information over longer periods of time, although, at the same time, they produce output only rarely. Our hypothesis is that using a percentage of slow neurons should increase the length of the fading memory and thus increase LSM performance. This percentage will probably be an important parameter, as using too many slow neurons will drastically reduce the activity in the network.

2. The enforcement of a hierarchical structure on the liquid. The column layout used by Maass et al. is simple but effective. In fact, columns such as these have been found to exist in the human brain as well. In the visual system, for example, researchers have found columns of neurons that correspond to similar stimuli. For example one column responds to stimuli to the left eye, another column responds to stimuli to the right eye. This is called ocular dominance and was discovered by Hubel and Wiesel [6]. These columns are clusters of neurons with strong connections to each other that connect only sparsely to other strongly connected regions. It is this last arrangement that we will model for our research. We will setup a number of columns of neurons, each of which is only connected to the previous column. Only the first column is connected to the input and only the last column is used for the readout. In this way, the input is processed in phases, where only the final phase is used for learning. We will also look at branching these phases, which could allow for successive columns to code for specific features in the input. Finally we also look at the parallel columns used by Maass et al.[1].

1.3 Research Method

The goal of our research is to improve LSM performance in general, however it is very hard, if not impossible, to mathematically prove any performance increases. Doing so would require using the separation and approximation properties, and showing that improving one or both increases liquid performance. As was shown by Kok in his thesis [3], an improvement in separation (measured using the Euclidian distance as posited in [1]) doesn't necessarily give any improvement in LSM performance. Hence, for our research we chose to test the effects of our changes by measuring LSM performance directly. We decided on a number of

useful and interesting tasks and apply both the original LSMs and our modified versions to them.

Maass et al. used a number of very simple artificial tasks to test the original LSMs on. Although we did test our modifications using these tasks, we felt they were too limited for a good evaluation. So in addition to their tasks, we used a number of real world problems for our research. This should give us tasks with sufficient complexity that our changes can yield actual improvements, if the tasks were easily solved by the original implementation, there wouldn't be much room for improvement. It also allows us to test the power of LSMs for real applications and gives us a better idea of the actual usefulness of the technique.

The additional tasks we will be looking at are the recognition and classification of spoken digits and the classification of piano music by different composers. We chose these particular tasks because of their real world relevance and the complexity of the time-series involved. Both tasks consist of fairly complex signals that need to be combined over relatively large time-frames to classify correctly. For example, it's hard to classify a word based on only the first sound that comprises it and this is even harder in the case of piano music, because all compositions use the same keys.

Apart from this, our choice of tasks was also influenced by the availability of samples. For the speech recognition task we had access to the very useful Corpus Gesproken Nederlands [7], which allows one to easily extract all manners of speech samples from their huge database of transcribed and annotated speech recordings. The music samples were even easier to acquire, as MIDI-files are often simply a digital transcription of the original sheet music, which in the case of Bach and Beethoven, the composers we used, is in the public domain and therefor freely available.

We will set up these tasks in such a way that the performance of unmodified LSMs is at an acceptable level. Then we will measure the difference in performance that our modifications to the LSMs make. For example, we will use a number of composers such that the baseline performance isn't so high that our modifications won't show any change in performance.

We will run both original LSMs and our modifications on these tasks and compare the results. Because LSMs involve a fair amount of randomness during generation, we will run these tests a large number of times to get a clear average of the performance.

1.3.1 Position of the topic in the broader field of AI

This research is interesting from the viewpoint of AI because it includes a number of different aspects. On one hand, spiking neural networks are strongly biologically inspired and the different structures we look at are also based on the biology of the brain.

On the other hand, LSMs are a cutting edge technique in the field of machine learning, because they combine a few different techniques that play a big role in AI. The sub-symbolic approach to machine intelligence that comes with such techniques also fits very well in a materialist, emergent view of human intelligence. Similar to how the complexity of all the biological and physical processes in the brain prevent us from pointing out the location of the mind, the complexity of the sub-symbolic approach makes it hard for us to figure out the details of how each part of LSMs contributes to the completion of the actual task.

We also feel our evaluation tasks are both interesting and should have a strong appeal to the AI community. However, both of these tasks have already been solved at a sufficient level to be applied in daily life. Speech recognition of course sees plenty of use, with large numbers of call centers using it. Google, for example, offers free speech-to-text translation of voice-mail messages through Google Voice [8].

Music recognition is available on all smart-phones from Shazam [9] and other vendors like SoundHound. Shazam in particular, is based on a very different technique than ours. It uses an enormous database of 'music fingerprints' to compare the sample you want to identify to. Although this is a great real-world solution, it has very little to do with AI. Our approach, with its liquid of Spiking Neurons is very much an application of AI and seeks to apply knowledge from biology and neurology to solving real world tasks.

Chapter 2

Machine Learning and LSMs

2.1 Machine Learning

Liquid state machines are a research subject within the field of Machine Learning, which is itself a part of the discipline of Artificial Intelligence. Machine learning is concerned with the automated improvement of machine performance based on available information. Machine learning can be divided into three types:

Supervised learning In supervised learning, we have a problem that we want to solve and we also have a number of solved cases of this problem. Because we know the correct answer for these cases, we can test our proposed solution and improve its performance by changing its parameters, in effect teaching it to perform better.

Mathematically speaking, we have a set of pairs of example data (x, y) with x in set X and y in set Y and the aim is to find a function $f : X \to Y$ that fits the known data. Because we only have a limited number of samples, we have to watch out that we don't overtrain our algorithm. This would cause it to lose generalization power, and make more mistakes on unseen samples (called overfitting).

Unsupervised learning In unsupervised learning we don't know the correct answer to any cases. Instead we have a so-called cost function to determine how right or wrong every possible answer is. We have to improve our solution by minimizing the cost function, thus increasing performance.

Mathematically speaking, we have some data x in set X and a cost function $C : XxY \rightarrow \mathbb{R}$. The goal is to find a function $f : X \rightarrow Y$ that minimizes the cost function.

Reinforcement learning Finally, there's the case where we don't get any data on the task beforehand. Usually the actual data is obtained through interaction with some kind of environment.

The best-known example of reinforcement learning is that of artificial agents in a simulation. The agents take actions, which help determine the cause of events, but often the exact influence of each action on the long-term results is unknown.

The tasks we will look at in this thesis are supervised learning classification tasks. We have a number of samples including their correct classifications and train our LSM to generalize from these samples to other unseen data.

2.2 Biological Neuron Models

Before we look at neural networks used in Machine Learning, we will give you a quick overview of Spiking Neuron Models in biology. We will start with a short introduction on the physiological and chemical properties of neurons in the human nervous system.

Neurons are much like other cells in the human body. They consist of a cell membrane containing all the normal cellular machinery, suspended in the intracellular fluid or cytoplasm. From this cellbody, or soma, extend the main distinguishing features of neurons, the axon and dendrites. These protrusions are extensions of the cell membrane and contain cytoplasm that is connected with that in the soma.

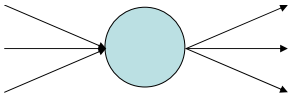


Figure 2.1: Schematic representation of a neuron

Neurons are connected to each other by these dendrites and axons, specifically an axon from one neuron connects to one or more dendrites (and sometimes the soma) of other neurons at sites called synapses. The pre-synaptic neuron is connected with its axon to the dendrite of the post-synaptic neuron. Neurons generally have a single axon and any number of dendrites, but both of these

can fork into many smaller branches. Axons can extend to enormous lengths compared to the neuron size, the motor neurons in the spinal cord, for example, can have axons that are a metre or more in length.

2.2.1 Resting potential

The intra and extra cellular fluid are watery solutions containing ions such as K^+, Na^+, Ca^{2+} and Cl^-. The cell membrane separating the fluids is permeable to water, but only selectively permeable to these ions. It contains so called non-gated ion-channels, that always allow certain ions to pass through. There are also gated channels that only allow ions through under the influence of electrical, chemical or physical stimuli. Finally, there are ion-pumps in the cell membrane, that actively pump ions into and out of the neuron.

The ion pumps continuously push Na^+ ions out of the neuron and K^+ ions into the neuron. This leads to a higher concentration of Na^+ outside and a higher concentration of K^+ inside. Under normal circumstances, chemical diffusion pushes ions from areas of higher concentration to areas of lower concentration. However, because the cell membrane is more permeable to K^+ (there are many more non-gated K^+ channels) than Na^+, the Na^+ ions can't flow back into the neuron, although K^+ ions can flow freely.

The net effect of this is that a lot of positively charged ions, both Na^+ and K^+, leave the neuron. This gives rise to a potential difference, between the relatively positively charged extra-cellular fluid and the relatively negatively charged intra-cellular fluid; this is called the membrane potential. A high membrane potential in turn causes a flow of positively charged ions into the neuron, consisting mostly of K^+ (again due to the number of channels). The potential at which the net flow of K^+ is 0 is called the equilibrium potential or resting potential of the neuron.

The exact equilibrium potential can be calculated separately for each ion using the Nernst equation, or for all ions together using Goldman's later generalization, now called the Goldman equation.

2.2.2 Action potential

When a synapse is activated, the pre-synaptic neuron releases neurotransmitters into the tiny gap between the pre- and post-synaptic neuron, called the synaptic cleft. These neurotransmitters bind to any free receptors on the dendrites of the post-synaptic neuron, causing additional gated ion channels to open. This allows ions to flow into the neuron, leading to an increase in membrane potential locally. The current this causes then spreads through the neuron, through electrotonic

(or passive) conduction. If the current is strong enough (on its own, or together with currents from other synapses on the same neuron), it will cause an action potential at the axon hillock (which is the site where the axon connects to its soma).

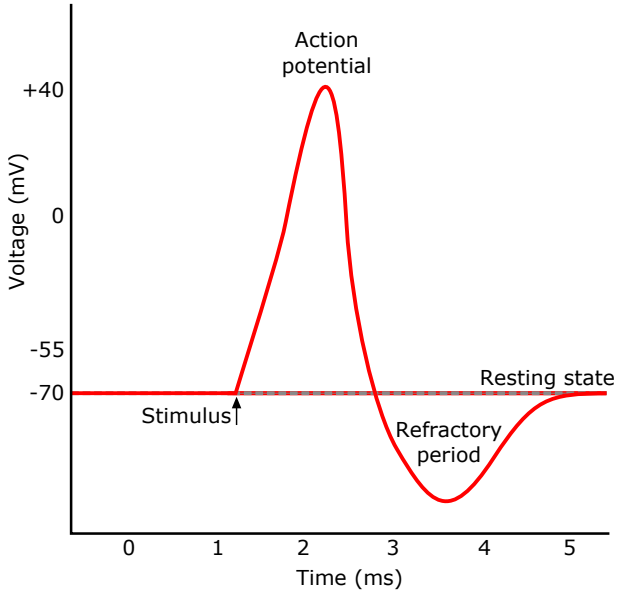


Figure 2.2: Action potential cycle

An action potential only happens if the current at the axon hillock is strong enough, crossing a value called the threshold. If this is the case, many more gated Na^+ channels open, strengthening the current and causing it to flow further down the axon. This process repeats itself at more sites along the axon, called nodes of Ranvier, all the way down the axon. At each of these nodes, more channels open and more current flows into the neuron. This leads to the characteristic peak in membrane potential shown in Figure 2.2. Any synapses that the current passes on the way are activated, possibly leading to further action potentials in their respective post-synaptic neurons.

This process of a neuron having an action potential, is also called the firing of the neuron.

2.2.3 Refractory period

At some point during the action potential, the membrane potential crosses the threshold for K^+ gates to open, allowing K^+ to flow out of the neuron quickly. This decreases the membrane potential, even below the resting potential, causing all gated channels to close. After this, the non-gated channels allow the neuron to

slowly return to its resting potential. This period of very low membrane potential, during which it is hard to trigger another action potential, is called the refractory period.

In fact, the gated Na^+ channels are completely inactive for a while after an action potential; this is called the absolute refractory period, because the neuron simply cannot generate an action potential during this time. The period following that until the resting potential is reached is called the relative refractory period and the neuron is able to generate an action potential, but only if the post-synaptic currents are much larger than normal.

2.2.4 Excitatory and inhibitory

The strength of the signal at a synapse can vary, causing the pre-synaptic neuron to release more or less neurotransmitter, resulting in a larger or smaller current in the post-synaptic neuron. It can take multiple action potentials from one or more neurons to cause an action potential in a post-synaptic neuron. When an action potential does occur, however, the amplitude is always the same; action potentials are all or nothing.

Some synapses can also cause an inverse current in the post-synaptic neuron, making it less likely to generate an action potential; these are called inhibitory post-synaptic potentials, as opposed to excitatory post-synaptic potentials.

2.2.5 Dynamic synapses

There are two additional factors influencing the strength of the post-synaptic current. On the post-synaptic side, it takes a while for the receptors to be freed of bound neurotransmitters and be able to cause another post-synaptic current. On the pre-synaptic side, a neuron only has a limited amount of neurotransmitter at the synapse, and this needs to be replenished by re-uptake of the neurotransmitters from the synaptic cleft.

2.2.6 The Hodgkin and Huxley model

Spiking neuron models, named after the characteristic spike shape of action potential graphs, describing in detail the dynamics of the above processes, have been around for a long time. Lapicque published his integrate-and-fire model in 1907 [10].

From the 1930's through the 1950's, Alan Hodgkin and Andrew Huxley performed extensive experiments on the giant neuron in squid [11]. This massive neuron,

that controls part of the squids water jet propulsion system, has probably evolved because of the increased speed with which action potentials travel along larger neurons due to decreased resistance.

Their experiments confirmed the results of Goldman's equation and lead to the publication in 1952 of a paper [11] containing a far more detailed model of the electrochemical processes in neurons than the ones available at that time.

Hodgkin and Huxley's model is still very relevant today, even if it has been improved since then, and it is the basis for the model we'll describe later in this chapter.

2.3 Artificial Neural Networks

Artificial neural networks (ANNs) are a computational technique based loosely on the biological neurons described above. An artificial neural network consists of a collection of interconnected neurons that individually do only simple transformations on data, but together can work as a universal function approximator [12].

The simpler ANNs, such as feed-forward neural networks (FNNs) and recurrent neural networks (RNNs), use a very basic neuron model. This model doesn't include action potentials, so we call it a non-spiking neuron model.

The more complicated ANNs such as spiking neural networks use more complex spiking neuron models. Some use the Hodgkin and Huxley model or a variation of it, most use a simplified spiking neuron model such as the leaky integrate and fire model (LI&F). This simplified model is still realistic enough to allow for complex behaviour, but it saves a lot of time when simulating a neural network.

In the next sections, these networks will be discussed in more detail.

2.3.1 Non-spiking Neural Networks

The non-spiking neuron model used in some artificial neural networks does away with action potentials and the other complex dynamics associated with the membrane potential. They consist of the following, biologically inspired, parts:

- A number of incoming connections, analogous to synapses on the dendrites;
- A number of outgoing connections, analogous to synapses on the axon;

- An activation value assigned to each neuron, analogous to a biological neurons' membrane potential.

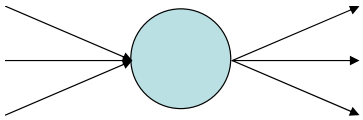


Figure 2.3: Artificial neuron

Each of the connections has a connection strength, determining the signal strength. This is similar to how synapses can be stronger or weaker in biological neurons. A common choice is to use a number in the domain [-1, 1] for the connection strength.

The activation value of a neuron is also a number in the domain [0, 1]. Unlike the membrane potential of spiking neuron models this doesn't have any complex dynamics. The activation can always be calculated by taking the sum of every 'pre-synaptic' neurons activation value times the connection strength to this neuron:

$$x = \sum_i w_i g_i \tag{2.1}$$

Where g_i is the output of a previous neuron and w_i is the weight of the connection from that neuron.

Because this function has a range greater than [0, 1] it has to be rescaled, which is done by the activation function. One such activation function that is commonly used is:

$$f(x) = \frac{1}{1 + e^x} \tag{2.2}$$

This is also known as the sigmoid function.

Feed-forward Neural Networks

One of the simplest forms of artificial neural networks is the feed-forward neural network (FNN). In this type of neural network, the network is made up of three types of layers:

- The input layer;
- The hidden layer(s);

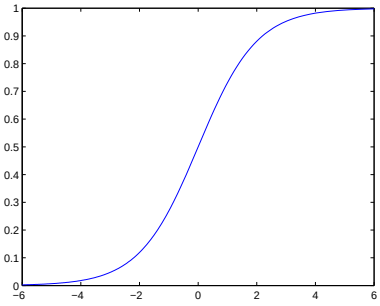


Figure 2.4: Sigmoid activation function

- The output layer.

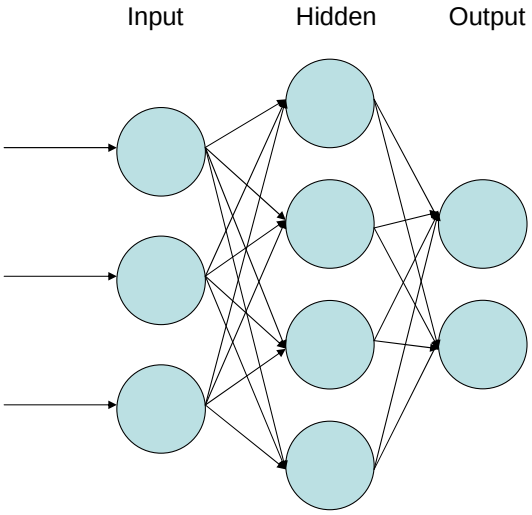


Figure 2.5: Feed-forward Neural Network

The neurons in the input layer have no incoming connections, but their activation value is determined entirely by the input. A FNN can contain any number of hidden layers. The neurons in the hidden layers each have incoming connections only from the previous layer and outgoing connections only to the next layer, hence the name feed-forward. The output layer is the last layer of a FNN and its activations are the output of the FNN.

Finally, all the neurons in the hidden and output layers are connected to the Bias Neuron. This is a neuron that always has an activation value of 1. It is necessary to include this neuron for the network to be able to approximate certain functions, like the logical function NOR.

The most common training algorithm for use with FNNs is back propagation. Back propagation is a supervised learning algorithm, so it uses a training set of samples for which the desired output is known. The difference between the output of the FNN that is being trained and the desired output is called the error. The goal of training is minimizing this error.

Because of the layered nature of FNNs, we need to look at the contribution to the error by each of the neurons in the network. To do this, we propagate the error backwards through the network, hence the name back propagation. This propagation works similarly to normal signal propagation in neural networks, but in the other direction. Where a neuron's activation value is the sum of the values of the neurons in the previous layer, the error contribution of a neuron is equal to the sum of the errors of the neurons in the next layer, multiplied by the weight of the connections to each of these neurons.

Once we know the contribution of every neuron to the error, we use gradient descent to minimize this error. The gradient, in this case, is the gradient of the contribution of a neuron to the error. In other words, if a higher activity in a neuron contributes positively to the total error, the gradient is positive; if it contributes negatively to the error the gradient is negative. In order to reduce the total error, the connections from this neuron to the next layer are adjusted by a factor eta (the learning speed) multiplied by the gradient.

Given the error function (where t is the expected output and o is the actual output of output-neuron j),

$$E = \frac{1}{2} \sum_{j \epsilon outputs} (t_j - o_j)^2 \tag{2.3}$$

the contribution to the error δ_j of a single output neuron is

$$\delta_j = \frac{1}{2}(t_j - o_j)^2 \tag{2.4}$$

for output-neurons and

$$\delta_h = \sum_{j \epsilon outputs} \delta_j w_{hj} \tag{2.5}$$

for hidden-neurons.

The gradient for a connection from neuron i to neuron j is then given by equation

$$\Delta w_{ij} = \delta_j \frac{df(x_j)}{dx_j} x_j \tag{2.6}$$

where $f(x)$ is the sigmoid activation function from 2.4 and x_j is the total input for neuron j.

Because we want to decrease the error we change the weights proportional to the negative of the gradient. In other words, we change the connections by the gradients multiplied by a factor $-\eta$, the learning speed.

$$w_{ij} = w_{ij} - \eta\delta_j \frac{df(x_j)}{dx_j} x_j \tag{2.7}$$

When using back-propagation to train a neural network we have two options for when we change the weights. One is called batch training, in which we use the sum of all gradients from an entire batch as the basis for changing the weights. The other is online training, where we use the gradients from a single sample for changing the weights. It's also possible to use something in between those two, like fixed size batches.

Recurrent Neural Networks

When solving a problem which involves time series data, like speech recognition or approximation of a moving average, it becomes necessary for an ANN to combine data over time. Because FNNs are stateless, they give a single output for a single input, the only way to have them combine data is to input it all at once, for example by presenting the data for multiple time steps at the same time. Alternatively, it is also possible to generalize our concept of FNNs by allowing for recurrent connections, incoming connections from a neuron in one layer to neurons in previous layers. A neural network including loops like this is called a Recurrent Neural Network (RNN).

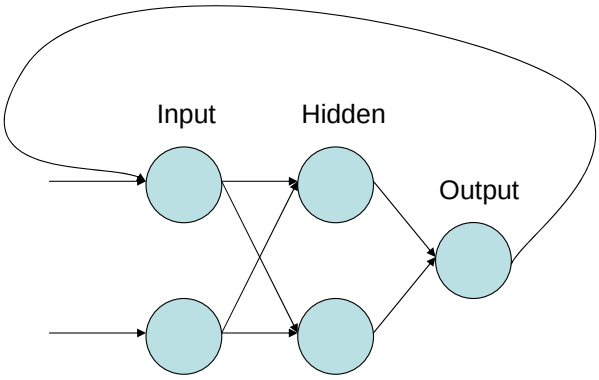


Figure 2.6: Recurrent Neural Network

This seemingly simple change has rather big consequences for the properties of the network. It is now suddenly possible for the network to keep an internal state, which allows us to tackle a whole new array of tasks, like computing the moving

average of a series of numbers (a task that can't be done without combining data from different steps). It does come with its fair share of problems, though. The biggest one is that training of RNNs is generally very slow.

To give an example, let's look at the generalization of back-propagation to RNNs: back-propagation-through-time. The algorithm is identical to what we discussed above, with one minor change. Because a neuron can now have a number of inputs from a previous time-step, it's necessary to look at the impact on the error of previous timesteps as well. This is best explained with an image

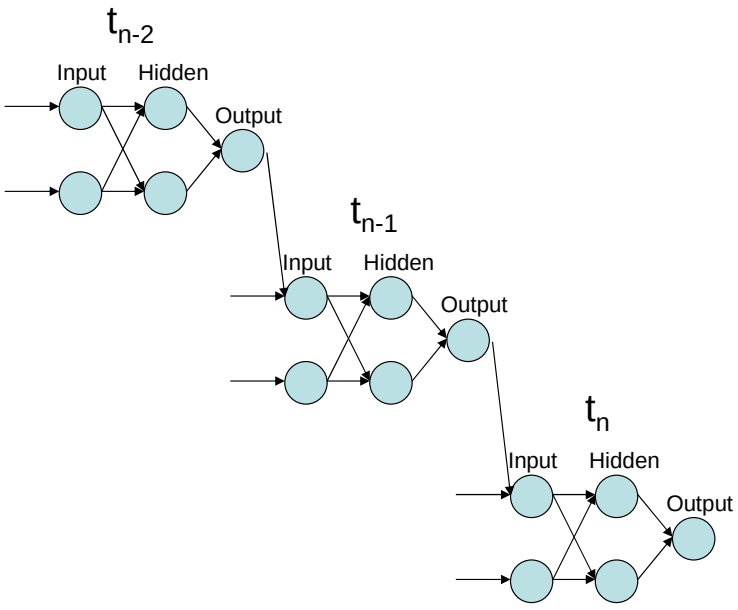


Figure 2.7: Backpropagation through time

These connections go all the way back to the first time-step, but in practice are usually limited to a fixed number of steps. Even then, the cost of minimizing the error by back-propagation for an RNN is significantly increased from the FNN case.

There are some other, more innovative approaches to training RNNs which we will not discuss here, but in general, training RNNs remains slow.

2.3.2 Spiking Neural Networks

As we mentioned, the non-spiking neuron model used in FNNs and RNNs is very simplified, leaving out such things as action potentials. The next model we

look at is much more complex, as it does model action potentials (also known as spikes). This changes both what the networks can do and how we use them.

These so-called Spiking Neural Networks are neural networks built from spiking neurons. Although fairly powerful, especially in dealing with time series data, SNN's have a number of problems. Both the input and the output of SNN's are in the form of spike trains, which are nothing more than a series of spikes. This embedding of information in spike trains is called spike coding. Due to their subsymbolic nature, the encoding and decoding of spike trains is a difficult task and there are many ways of looking at it, based on observations from neurobiology.

For example, rate coding is a very simple model that uses the frequency of spike trains as its main information carrier. Temporal coding on the other hand, is any kind of coding that uses the precise timing of individual spikes as its basis. There are so many ways to look at spike timing alone, that temporal coding is a research subject of its own.

The other big issue with SNN's is similar to the training issue with recurrent neural networks, as many SNN's use recurrent connections as well. Training SNN is hard and very time-consuming. One of the most well-known training algorithms for SNNs is in fact an adaption of back-propagation-in-time to spiking neural networks.

Leaky Integrate and Fire Neuron

While any spiking neurons will do for SNNs, we will look at one particular model here, the Leaky Integrate & Fire [13, 14] model. While it is still a gross simplification compared to the Hodgkin and Huxley model, it allows for very dynamic behaviour. It is also a lot faster to compute than the Hodgkin and Huxley model, allowing us to simulate larger networks without much of a slowdown.

Like in the non-spiking neuron model, neurons consist of the following parts:

- A number of incoming connections, analogous to synapses on the dendrites;
- A number of outgoing connections, analogous to synapses on the axon;
- An activation value, analogous to a biological neurons' membrane potential.

Each of the connections has a connection strength in the domain [0, 1]. But where negative connection weights were used previously, we now have inhibitory neurons.

Unlike the non-spiking model, the activation value in a neuron isn't simply calculated from the activation values of all the neurons it has connections from. Instead, the activation value is more like the biological membrane potential. The activation value of a neuron normally starts at the resting value, similar to the resting potential and when the neuron receives a spike, the activation value is raised.

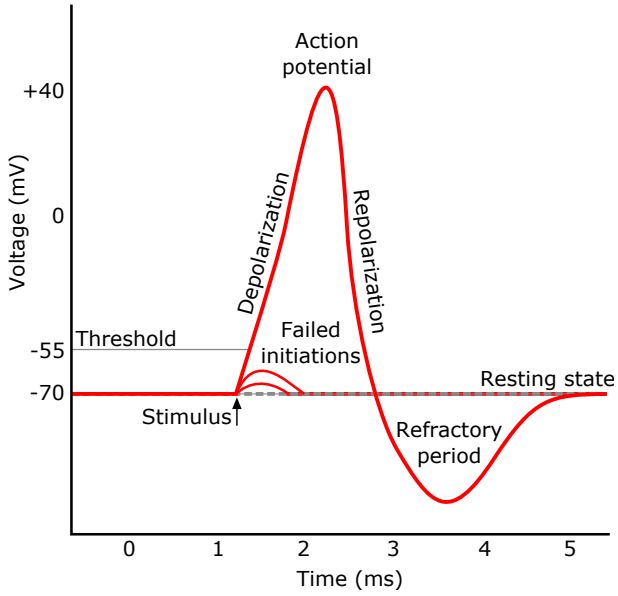


Figure 2.8: Action potential

Whenever the activation value crosses the firing threshold, the neuron fires, sending a signal to all neurons it has connections to. After firing, the neuron observes a relative refractory period: the activation value is reset to a value lower than its resting potential, the reset value. LI&F-neurons can also have a absolute refractory period, which is a small period during which the activation value is fixed to the reset value. This process is comparable to the biological neuron described in Section 2.2.

Whenever the activation value of a neuron is either higher or lower than its resting potential, its activation value slowly increases or decreases towards the resting potential. This is where the leaky part in the name comes from (differentiating it from Lapicques Integrate & Fire model).

The membrane potential u at step $i+1$ is given by

$$u_{i+1} = u_i + \frac{(E + R \cdot I_i - u_i)}{\tau \cdot \Delta t} = u_i + \frac{(E + R \cdot I_i - u_i)}{R \cdot C \cdot \Delta t} \tag{2.8}$$

where E is the membrane resting potential, R is the membrane resistance, I is the input current, τ is the membrane time constant, C is the membrane capacitance and Δt is the elapsed time from i to $i + 1$.

Dynamic Synapses

Some neuron models also model the dynamics of biological synapses. During an action-potential, the pre-synaptic cell emits a quantity of neurotransmitter which binds to receptors on the post-synaptic cell. The pre-synaptic cell can only emit so much neurotransmitter and the post-synaptic cell can only bind a limited amount of it. This mechanism creates a dynamic where subsequent action potentials have less influence on the post-synaptic cell until the synapse gets some time to recover. More information about using dynamic synapses can be found in Markram 1998 [15].

2.4 Time Series Problems

Time series problems are a class of problems where the input consists of series of data-points measured or calculated at regular time-intervals. Examples of time series are speech, music and other sound samples, moving images and stock market histories.

By their very nature time-series problems require the collection and combination of information over a span of time to solve. Hearing the first few notes of a piece of music or the first milliseconds of a spoken word isn't always going to provide us with enough information to recognize them. Looking at a single screen capture from a movie doesn't always let us identify it and a single stock quote won't let us recognize or predict market trends. Although there are other machine-learning techniques out there that can solve time-series problems (such as Hidden Markov Models, see Rabiners 1990 paper [16]), only the Artificial Neural Network solutions based on the techniques mentioned above are relevant to our research.

Of the techniques mentioned, FNNs are the most ill-suited to time-series problems. This is because FNNs are completely stateless, they take a single input and give an output for that. Time series problems aren't usually looked at in this way, although they can be adapted to this. Suffice to say, that any such solutions are far from ideal.

Instead we can look at RNNs, as these have a state, which is passed from each time-step to the next. As we looked at above, training RNNs is somewhat problematic because of the recurrent connections. In order to perform well on highly

dynamic problems a large number of time-steps need to be used for the backpropagation through time algorithm. This is not improved by the difficulty of some time-series problems that require a lot of state to be kept and as such require many recurrent connections and an increasingly large network.

This is partially solved by using a SNN, which keeps a separate internal state for each of the spiking neurons in the network. Although this cuts down on the complexity of the network, it does no such thing for the complexity of the training. Due to the dynamic and non-linear nature of SNNs it is very hard to calculate the contribution of every individual neuron to the total error of the SNN during training. There are some algorithms that do this, but they are both mathematically complex and computationally intensive.

The techniques available to us at this point, simply aren't good for time-series problems or are very time consuming. This is likely what lead to the creation of Liquid State Machines by Maass et al.[1].

2.5 Liquid State Machines

We have already quickly explained the basics of LSMs in the introduction, but in this chapter we will look at them in more detail. In a sense LSMs combine the power of the above techniques. SNNs, which are inherently powerful due to their dynamic and non-linear nature, for the liquid, and the simplicity of training of FNNs, for the readout.

2.5.1 Liquid

The liquid is the part of the network that the input goes into. It is made up of a large number of spiking LI&F-neurons, located in a virtual three-dimensional column (in the case of Maass et al.), of 3 by 3 by 15 neurons. The liquid has two important functions in the classification of time-series data.

First, its fading memory is responsible for collecting and integrating the input signal over time. Each of the neurons in the liquid keeps its own state, which already gives the liquid a strong fading memory. The activity in the network, the actual firing of the neurons, can also last for a while after the signal has ended, which is another form of memory.

Second, it's in the liquid that different input signals are separated, allowing for the readout to classify them. This separation is hypothesized to happen by increasing the dimensionality of the signal. If the input signal has 20 input channels, this is transformed into 135 (3 * 3 * 15) signals and states of the neurons in the liquid.

The first step in the creation of a liquid is the creation of all the individual neurons. The liquids proposed by Maass et al. contain both small and large variations in the neurons used.

The largest variation is the use of both excitatory and inhibitory neurons. In short, excitatory neurons that fire increase the membrane potential of their post-synaptic neurons, making them more likely to fire, exciting them. Inhibitory neurons on the other hand decrease the membrane potential, making the post-synaptic neurons less likely to fire, inhibiting them. Maass et al. use liquids with 20% randomly selected inhibitory neurons.

The smaller variations are in the different parameters of the LIF-neurons. In particular they use a small range of variation in the initial and reset potential of the neurons.

The second step is creating the connections to and within the liquid. First the connections from the input to the liquid are made, for every pair of input channel and liquid neuron there is a flat 30% chance of there being a connection.

Second the connections within the liquid itself are made. For each pair of neurons $< a, b >$ there is a random chance of a connection, according to equation 2.9.

$$P_{connect} = C \cdot e^{-(D(a,b)/\lambda)^2} \tag{2.9}$$

Here C is a constant that depends on the type of both neurons. Maass et al. use .3 for excitatory-excitatory connections, .2 for inhibitory-excitatory connections, .4 for excitatory-inhibitory connections and .1 for inhibitory-inhibitory connections.

$D(a, b)$ is the euclidian distance between neuron a and b and λ is a parameter that influences how often neurons are connected and the distance these connections bridge on average.

This is why the neurons have a spatial placement in the column, influencing the connections between them. Maass et al. showed that the performance of an LSM was greatly influenced by the choice of λ. At 0 there are no recurrent connections within the liquid, which decreased performance significantly, showing the importance of these connections. With a high value of λ the performance also decreased, which Maass et al. suppose is due to to many long distance connections in the liquid leading to increasingly chaotic activity. There is anatomical research of the cortex that suggests that most synaptic contacts of cortical neurons are within the same cortical area (see Tsodyks et al.[17]), which supports this theory.

2.5.2 Readout

The readout is one or more FNNs that use the activity of the liquid to approximate some function. In our tasks this is a classification function. We have one readout network for each of the classes we are trying to separate, and each of these readouts is trained to give a high output for samples of the class it should recognize and a low output for samples from all the other classes.

The inputs for the readout networks are so-called readout-moments. These are snapshots of the liquid activity taken at a regular interval. The readout can include membrane potentials or spikes or both. Whatever measure is used, the readout represents the state of the liquid at some point in time.

The readout networks use each of these readout-moments to make a classification of the sample and we take the classification that has occurred the most as the final classification for a sample. Classification is of course only one of the possible functions that can be approximated by the readout networks.

The training of these readout networks is done using the regular methods for FNNs, mainly backpropagation. There are also some variations on backpropagation such as Levenberg-Marquardt, which Matlab recommends for speed and performance.

2.5.3 Multi-tasking

One of the interesting properties of LSMs is the possibility of easily running multiple tasks on the same input in parallel. Using a single liquid and multiple readout networks it is possible to approximate multiple functions at the same time.

For example, the experiment in Maass et al. uses random pulse trains as input and has 6 different readouts for 6 different tasks which are performed in parallel. The tasks are sum of rates over the past 30ms, sum of rates over the past 200ms, detection of a spatiotemporal pattern, detection of a switch in spatial distribution of rates (they put the pattern and rate switch into their inputs deliberately) and spike coincidence over the last 75ms for two different pairs of channels. All of these are performed at the same time on the same liquid activity.

2.5.4 Online results and any-time algorithms

Another interesting feature of LSMs is the online nature of its results. If one records the readout-moments from the liquid as the input is being processed, it is possible to have the readout process them immediately. This way, one can have

partial results during the processing of the input, i.e. online. It is even possible to stop the algorithm at any point in time and settle for the approximations it has made up to that point. Hence for classification tasks and the like, LSMs can work as an anytime algorithm.

2.5.5 Universal computational power for LSMs

Maass et al. have shown that LSMs have universal computational power for computations with fading memory on functions of time. Providing the liquid satisfies the point-wise separation property and the readout satisfies the approximation property (as discussed in Section 1.1.1). The details on this can be found in the appendix to the article by Maass et al.[1].

2.6 Network structures

The group of stimuli a cortical neuron responds to is called the receptive field, which can be an orientation or position in visual stimuli, a location or range of frequencies in auditory stimuli or other groupings of similar stimuli. Research of the cerebral cortex has shown the existence of cortical columns, columns of neurons with very similar receptive fields. Groups of columns sensitive to different orientations or positions together make up the visual cortex.

LSMs already use a column layout for the liquid and using multiple columns independently connected to the input (or an intermediate layer of columns) could lead to differences in receptive fields between the different columns. Here it is unfortunate that our liquids are randomly generated, because this severely limits the specialization different columns can achieve. Still, having more neurons in the liquid should help increase the separation and in turn the LSM performance, while separating them into interconnected columns instead of a single large column prevents an explosion of activity in the column.

We have devised four ways of ordering and interconnecting the additional columns in the liquid, each somewhat inspired by biology. The two most obvious ones are parallel and serial columns.

Parallel columns In the parallel case (see Figure 2.9), the columns are each connected to the input and the activity of all the columns gets combined into readout moments. This form simply ups the number of neurons in the liquid, but due to the lack of connections between the columns it doesn't do much more. Maass et al. also discussed this variant and showed that it increases the computational

power of an LSM. The closest biological comparison is with redundancy in auditive or visual processing, where multiple cortical columns respond to similar input. The biggest problem with this ordering is that the readouts become very large and this can potentially slow down computations.

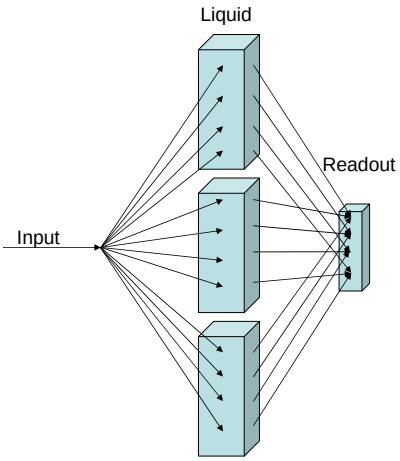


Figure 2.9: Parallel Liquid

Serial columns In the serial case (see Figure 2.10), only the first column is connected to the input and each successive column is connected to its predecessor. Only the activity of the last column is used by the readout, limiting the readout to the same size as the original liquids.

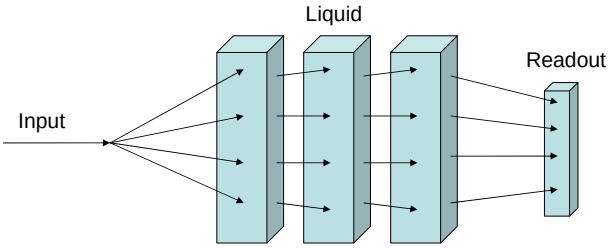


Figure 2.10: Serial Liquid

Biologically, this could be compared to the chained auditory or visual cortical areas, although that is a gross simplification, as we also know that their connections are neither one-way or serial. In Figure 2.11, a representation of pathways in the Macaque visual cortex, one can see the connected nature of its cortical areas. The problem with the serial approach is that the activity level in each successive

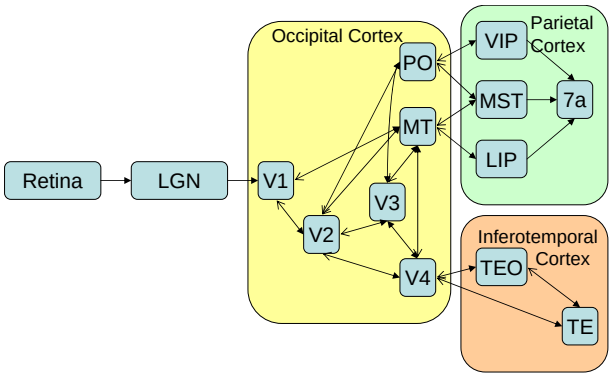


Figure 2.11: Pathways in the Macaque Visual Cortex

column tends to drop, leading to lower performance when using more than a few columns, because to much information is lost. This effect can be somewhat lessened by increasing the number of connections between adjacent columns. This is a sensitive parameter, however, as it can easily lead to exploding activity and loss of performance.

Our remaining two approaches are hierarchical in nature. Rather than having a number of lined-up columns, we use a layered structured. The first layer has a single column, the second layer has two columns, the third layer three, and so forth. Every column is connected to all the columns in the previous layer and again only the first layer is connected to the input and the last to the output. The resulting structure can be oriented in two different ways, the single column layer can be towards the input, which we call left-facing, or towards the output, which we call right-facing.

Right-facing structure The right-facing version (see Figure 2.12) can be seen as a bottom-up approach, where the left-most layer (the one containing the most columns) does low-level processing and the right-most layer does high-level processing. In the auditory system these could be the hair-cells up to the auditory cortex, or in the visual system this could be columns representing different receptive fields up to higher visual areas where the information is combined. The advantage to this approach is that the larger number of columns at the input lead to diverse activity and the readout is still done from a single column. However, the large amount of input of each successive layer could lead to exploding activity, so the connections between the layers need to be carefully tuned.

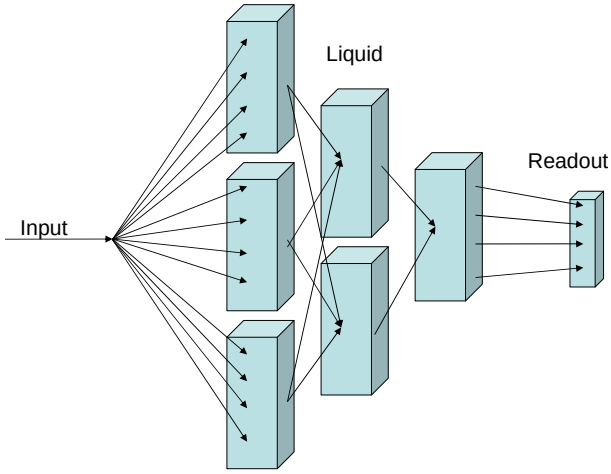


Figure 2.12: Right Liquid

Left-facing structure The left-facing version (see Figure 2.13) can be seen as a top-down approach, where the (many-column) output layer represents different aspects of the input and the (single-column) input layer represents basic signal processing. The advantage here is that there is more information for the readout to use, and the lower number of columns towards the input means less exploding activity in the later layers. The large readouts do lead to longer processing, though.

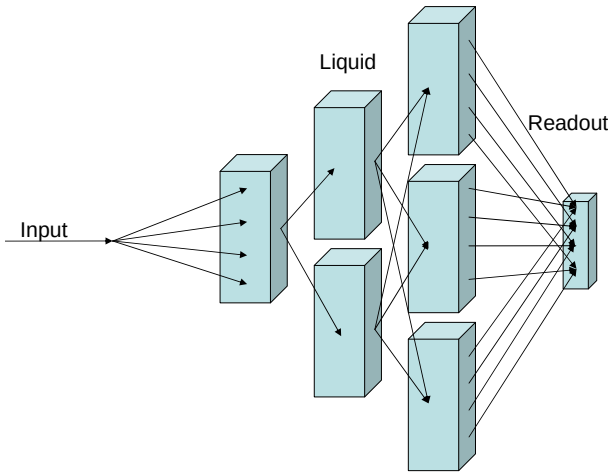


Figure 2.13: Left Liquid

Chapter 3

The three evaluation tasks and their samples

In this chapter we will look more closely at the three tasks we use to evaluate the performance of our changes to LSMs. Our three tasks consist of two existing tasks and one original one. The two existing ones are the tasks of Maass et al. ([1]) and Leo Pape's music recognition task [18]. The new task is a speech recognition task, based on samples from the Corpus Gesproken Nederlands [7].

It should be noted that Maass' task is fairly artificial, while we really wanted to use real world tasks. We decided to use it in addition to the other tasks to increase the amount of evaluation data.

3.1 Maass' task

This is the task that Maass et al. used in their original publication on LSMs. Maass' task consists of 6 subtasks using 6 concurrent readouts on one liquid. The liquid gets random pulse-trains as input. The pulse trains are generated with a varying pulse rate, according to:

$$r(t) = A + B\sin(2\pi ft + \alpha) \tag{3.1}$$

For training samples A, B and f are randomly drawn from the following intervals: A [0Hz, 30Hz] and [70Hz, 100Hz], B [0Hz, 30Hz] and [70Hz, 100Hz], f[0.5Hz, 1Hz] and [3Hz, 5Hz] and the phase α is fixed at 0°. For test samples, different values

are used, A 50Hz, B 50Hz, f 2Hz and α 180°, in order to show the generalization capabilities of LSMs.

Maass et al. also included two events in each of the samples, which are used for subtasks three and four. The events are a specific firing pattern at a random point in the pulse train and a rate switch at a random point in the pulse train.

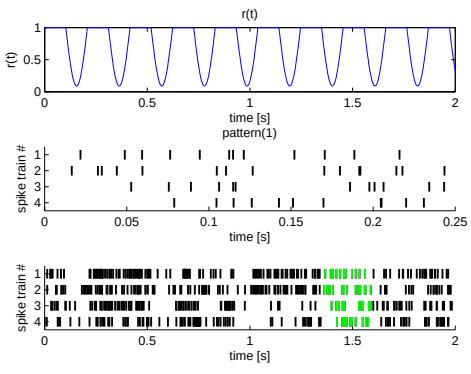
Maass' subtasks are as follows:

- Sum of rates of 30ms
- Sum of rates of 200ms
- Event detection for a specific firing pattern (didn't work)
- Event detection for rate switch (didn't work)
- Spike correlation between inputs 1 and 3
- Spike correlation between inputs 2 and 4

Because they didn't work in the version of the Learning-Tool we downloaded off of Maass' website, we removed subtasks three and four for our experiments.

Figure 3.1a shows the input used by Maass et al. At the top is the varying firing rate, as described by Equation 3.1. In the middle you see the patterns generated using this firing rate. Maass et al. use 4-channel input, so there are 4 patterns here. At the bottom is the final input, which has the patterns from the middle with jitter and noise applied. The highlighted part is one of the events mentioned above.

(a) Liquid input



(b) Liquid response

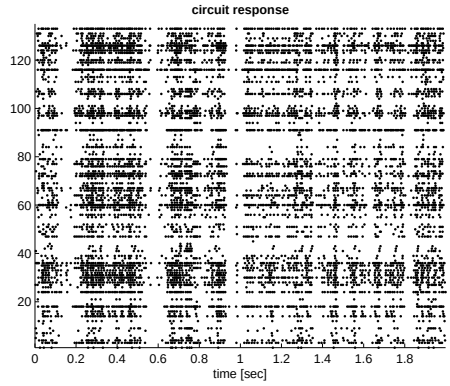


Figure 3.1: Liquid input and response for Maass' tasks

Figure 3.1b shows the resulting activity after inputting 3.1a to a randomly generated liquid. In Figure 3.2 taken from Maass et al.[1], you can see the results of training an LSM on the six tasks. The dashed lines are the target functions while the solid lines are the actual LSM output. At the left side of the image you'll also notice that Maass et al. use two parallel columns as we mentioned earlier.

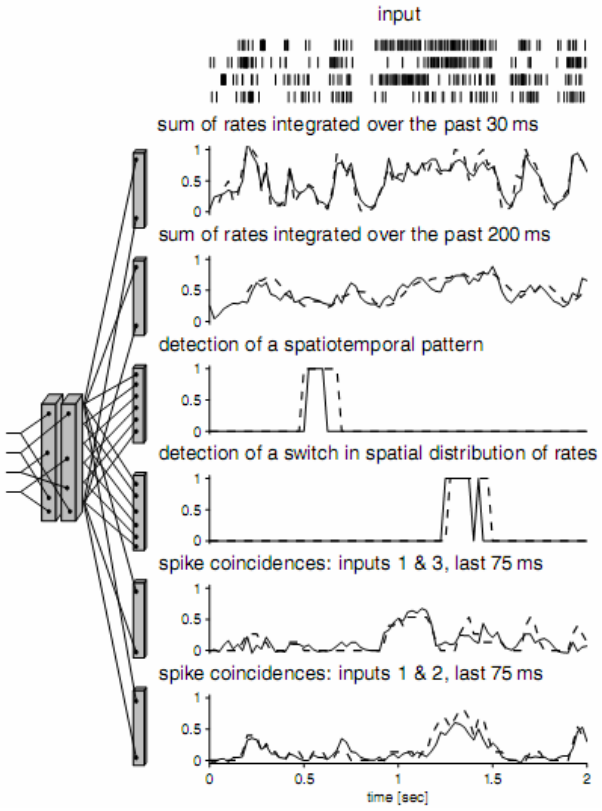


Figure 3.2: Plots of Maass' tasks taken from [1]

3.2 Speech recognition task

The second task is a speech recognition task. The samples are recordings of the digits 0 through 9 taken from the Corpus Gesproken Nederlands (CGN). The CGN is a large body of speech recordings from everyday sources like radio, telephone and television broadcasts. The content of these recordings has been transcribed and annotated. There are annotations on different levels, for example part-of-speech tagging, phonetic transcription and syntactic and prosodic annotation. Because of the annotations it is possible to automate the extraction of our samples from the larger speech fragments, which allowed us to use a relatively large number of samples.

Care was taken to limit the samples to similar pronunciations of the digits. This was done by removing samples from dialects and varying pronunciations, based on phonetic transcription. This reduces unpredictable variations in the difficulty of the task.

It was also necessary to divide the samples by sampling rate. Part of the samples used a 8kHz sampling rate and part of them used a 16kHz sampling rate. The 16kHz samples were used, because of the higher quality and the amount of available samples.

In this way we were able to acquire a fair number of samples of the digits 0 through 9. Unfortunately, due to the everyday nature of the CGN, not all digits were represented in equal measure, so our experiment will be limited by the lowest number of samples available for a single digit, which is 43 samples of the digit 9, as training on an unequal number of samples for the different classes could lead to skewed results.

3.2.1 Using speech samples: Fast Fourier Transform

The samples taken from the CGN were all wav files. A wav file contains nothing more than a single channel of amplitude. For our experiment we need multichannel data, because a single channel of input leads to only very localized activity in the liquid.

In human hearing, the signal that arrives at the ear, is also a single channel of amplitude data. Specifically the varying air pressure that is the physical nature of sound waves. In the inner ear, the hair cells transform this into a multi-channel signal, because different clusters of hair cells are sensitive to different frequencies.

We essentially do the same with our samples, by transforming our input into a multi-channel signal using Fast Fourier Transformations (FFTs). The result of a FFT is a number of channels corresponding to different frequency ranges.

The FFT is an algorithm capable of transforming data from the time domain to the frequency domain. Our original samples are in the time domain, because they contain a value for the signal amplitude that changes over time. The FFT algorithm transforms the signal to the frequency domain. The result is a list of frequencies and their respective amplitudes, also called a spectrum. The FFT gives us a single frequency spectrum for the whole sample, while we want the changes in the frequency spectrum over time. To do this, we use the FFT repeatedly, taking overlapping slices from the original sample, this is called a moving window.

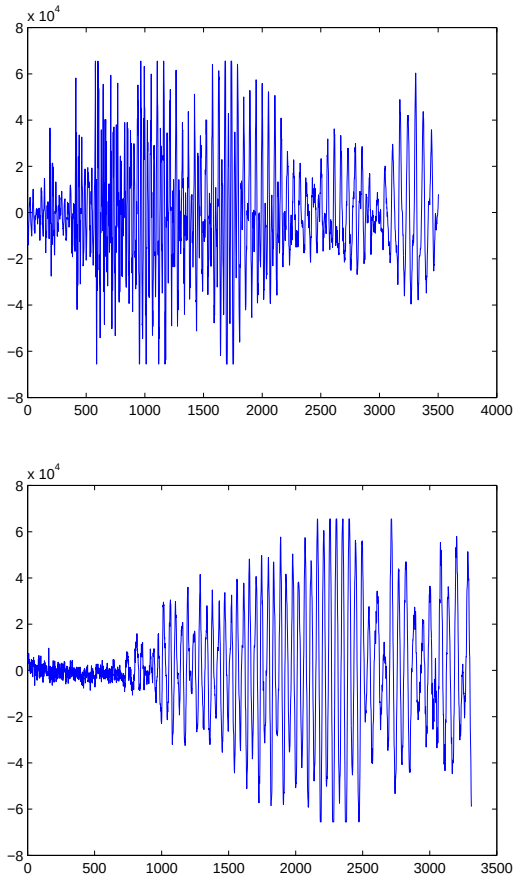


Figure 3.3: Waveforms for a sample for 'een' and 'twee' ('one' and 'two' in Dutch). The Y-axis is amplitude, the X-axis is the sample number at a sample rate of 16kHz

Because taking a rectangular window can lead to poor data, we make use of the well known Hamming Window. The Hamming Window is a filter that is commonly used when digitizing signals involving speech and other tasks involving narrow-band signals, because it helps reduce a phenomenon called spectral leakage. Combining the results of applying the FFT algorithm on each of these windows, we can construct the amplitude changes over time for a list of frequencies.

Due to how the FFT algorithm works, the frequencies in the results are grouped into a number of bins, corresponding to ranges of frequencies. Our choice of bins, however, was purely a result of the used FFT algorithm. It would be interesting to see how choosing these bins differently (for example based on their relevance to speech recognition or based on parallels in biology) might affect the speech

recognition task. A good starting point would be dividing the bins according to the mel scale (see the 1937 paper by Stevens et al.[19]).

3.3 Composer recognition task

The third task is the composer recognition task. When choosing this task, one of the things we wanted to do, was avoid using FFTs. To this end, we looked for a task that already had multi-channel input in a format that was easily adjusted to our sample requirements.

One thing that quickly came up was the idea of using piano music, however, because using actual recordings would again force us to use FFTs, we looked at using sheet music. Fortunately, there's a digital music standard that represents music in a format comparable to sheet music: MIDI, the Musical Instrument Digital Interface.

An earlier student of Marco Wiering, Leo Pape, had already worked with such a task, comparing the effectiveness of LSMs and Helmholtz Machines on the classification of piano pieces from different composers [18]. He used MIDI-files, because they are both easy to obtain (for music in the public domain) and they contain a clean representation of the composition that can easily be used as a multi-channel time-series input data.

For our experiment we have used the same set of samples that Pape used, so we have a good idea of the kind of performance we can expect on this task based on his results. The composers used are Bach and Beethoven (with Haydn and Mozart in reserve), specifically Book II from Bachs Well-Tempered Clavier and 30 movements from Beethovens Sonatas 16-31.

3.3.1 Using music samples: MIDI files

A MIDI-file is basically a long list of musical events, each with a relative time since the last event. Events can be key-presses or -releases, tempo-changes, aftertouch, pedal-presses, instrument changes etc.

For our purposes we were only interested in the key-events and the tempo changes. The key-events, press and release, together with the tempo give us all the information on the notes that make up the composition. We used both our own MIDI parser and an existing library by Tuomas Eerola and Petri Toiviainen [20].

By splitting the notes from the midi-file, each of the 128 tones gets its own channel. The result is a perfect example of multi-channel time-series data. These

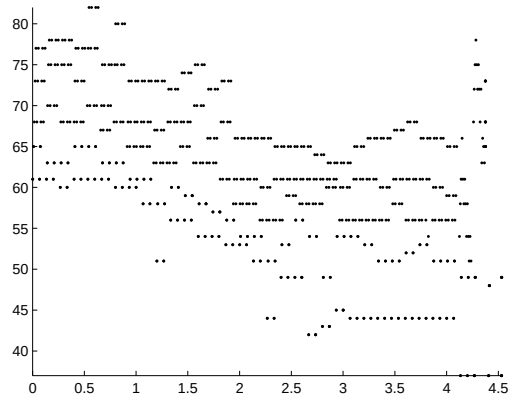


Figure 3.4: Input for music recognition task from Bach's Prelude in C-minor

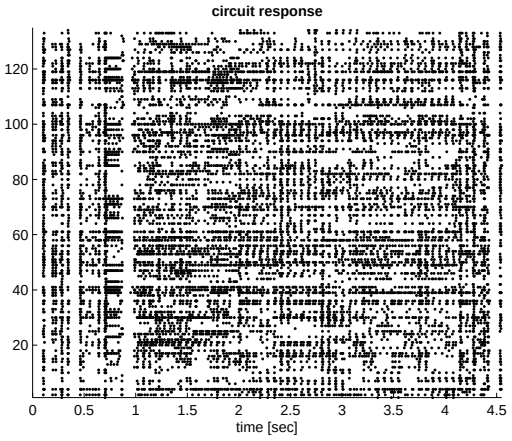


Figure 3.5: Liquid response for input 3.4

channels are then randomly mapped to 128 of the neurons in the liquid.

For more details we refer you to the official MIDI specification [21], or one of the many other explanations of the file-format on the internet.

We used the midi event timings, although we accelerated the pieces by a factor 30. This was done to get at least a certain amount of energy in the signal (too little and there would be hardly any activity in the liquid, too much and it would explode) and to supply the LSM with a good size sample to recognize.

Because we were generating events only for the key-pressed midi events, we also had to make a provision for longer notes. Leo Pape simply repeated these notes

for every time step during which they were held. We did something similar by taking the interval of the readout moments and repeating the notes for every readout interval that they lasted. One alternative might have been to generate key-release events as well, but this would add less energy to the signal for longer notes and, we felt, would depart even more from the original composition.

Both sets of samples (Bach and Beethoven) were randomly divided into a training set and a test set. The test set was chosen to be 20% of all samples. This gives us a good size test set, but still allows most samples to be used for training. A large training set was needed because the task isn't easy.

Chapter 4

LSM Software Implementations

When we started the research for this thesis we decided that it would be both educational and useful to develop our own implementation of LSMs.

On the one hand, writing our own implementation would force us to understand LSMs to a degree that we otherwise might not have. We had to look at every little piece of the algorithm and at every variable, in order to make the right choices.

On the other hand, it would give us the opportunity to easily try different neuronal and structural changes in an attempt to improve performance. This would also be possible with an existing implementation that comes with the source-code. But using existing code would require diving into this code and figuring out how it worked.

We had two existing implementations of LSMs, one in the Java programming language written by Leo Pape for his thesis [18] and one in Matlab [1]. The Java implementation didn't work out of the box. Reconstructing his code was less interesting than writing our own code. The Matlab version is the software Maass et al. developed for the research underlying their original paper.

4.1 Time steps or event-based

There are two approaches to implementing a Spiking Neural Network (what we use for the liquid): Event-based and time steps.

4.1.1 Time steps

The easiest to understand is the time steps. In this implementation, time is discretized into even steps. Time step size must be chosen based on the problem at hand, but between 1ms and 10ms is a good starting point. For every step the membrane potential of every neuron in the network is calculated, based on the input signal and the membrane potential and incoming spikes from the previous time step. If the membrane potential of any neuron is now over the firing threshold it fires, which will be used in the calculations of the next time step.

4.1.2 Event-based

An event-based implementation works quite differently. Instead of step-wise simulation of all the neurons, only the changes following an event are calculated. An event in this case means a spike in the input signal or an incoming spike from one of the neurons in the network. In order to know what to calculate next, there is a list of upcoming events. At the start of the simulation, this list only contains the events in the input signal.

The events are handled in chronological order, removing them from the list when they have been handled. The first event calculated, is the earliest event in the input signal. All the neurons affected by this event are then updated and it is calculated if and when any of them will fire. If a neuron is going to fire, a new event is added to the event list, at its calculated firing time. After all affected neurons are processed, the next earliest event is taken from the list the process is repeated.

This method of simulating a spiking neural network has two large advantages:

- The only limit on the precision of spike timing is the precision of the data type used to store event timings. This is useful, to make the simulation as accurate as needed. It also eases the use of some types of input signal that use events, instead of discretized measurements, such as MIDI.
- A lot fewer neuron updates have to be calculated, because only events cause updates. A readout moment is considered an event as well, which requires all neurons for which the membrane potential is required to be updated.

Two important disadvantages are:

- The mathematics involved are rather complex. Where a discretized simulation is fairly easy to implement, an event-based implementation requires

some advanced mathematics to compute upcoming events. A single neuron update also takes more time, because of these computations.

- It is hard to parallelize event-based simulations. Time steps can easily be calculated in parallel: Neurons can be updated independently of each other, so the work can be split as many ways as there are neurons. For event-based simulations only the updates that follow from a single event can be parallelized.

4.2 C♯ implementation

The C♯ implementation was started with the basics: the neurons. The neuron-model used is the Leaky Integrate & Fire model as described in Spiking Neuron Models by Gerstner and Kistler [14]. These neurons were relatively easily implemented and in preliminary tests behaved as expected. This testing was done with both Poisson generated Discrete Pulse Trains and Discrete Pulse Trains in the style of Maass et al, with a varying pulse rate. More information on the Poisson spike trains used, can be found in a handout by Professor Heeger [22].

Next up was the Liquid using the LI&F neurons we had developed. The basic concept setup of a liquid is simple: a fixed number of neurons regularly spaced on a 3d grid with random connections between neurons based on their distance. It soon became clear, however, that there are a lot of parameters that influence the stability of the activity in the liquid. If any of the connection strength, likelihood of connections, spike strength or the membrane time constant (a combination of the membrane resistance and capacitance) is off by more than a little, the liquid will show either exploding activity or almost no activity at all.

Another factor is the input signal, specifically the amount of energy the signal contains (a signal with a lot of spikes contains a lot of energy and vice versa). A high or low energy signal is likely to lead to exploding or no activity, unless the liquid parameters are changed to match it, even if for average signals the parameters work fine.

Initially, our liquid parameters were based on testing performed with the Poisson generated Discrete Pulse Trains and those in the style of Maass, and the liquid responses were as expected. However, the samples from our speech recognition task caused the liquid activity to explode, removing any chance of getting good information from the liquid response.

Getting the desired liquid activity by changing the parameters is a difficult process, done mainly by trial and error. Furthermore, debugging the code and behaviour of the liquid is virtually impossible due to the black box nature of LSMs.

Instead we had to settle for debugging single neuron behaviour and using plots of the liquid activity to see whether the liquid was working alright.

Due to the complexity of this process and the time-constraints of the research we decided to replace our own implementation with the pre-existing Matlab implementation by Maass et al. [23].

4.3 Matlab Learning-Tool

It should be noted that the version of the Matlab Learning-Tool on Maass and Natschlägers website [23] is apparently incompatible with the latest version of Matlab or it is an earlier version of their software, than they based their paper on. Two of the tasks they show results for in their paper (the 'event detection' tasks) don't work and needed to be removed before the software would run at all.

Using this Matlab implementation had the strong benefit that our results could be directly compared with those of Maass et al, because we used the same LSM implementation. In fact, we tested our neuronal variation (slow neurons) on the same tasks that they described in their paper. This way we could quickly test the effects of our liquid variations on the example tasks they used, giving us a clear indication of how well they were doing.

Furthermore, the Matlab implementation uses some tricks derived from event-based implementations, to make the time-based implementation run faster. Specifically, neurons that don't receive any input for a while are put to 'sleep'. Their membrane potential isn't updated, because there aren't any relevant changes, allowing us to update only a (sometimes small) subset of the neurons in the liquid and get significant speedups.

4.4 Modifications to the Learning-Tool

Two changes were made to the Learning-Tool for our research. The first thing we did was to include the slow neurons proposed in our research question. The second change made was repeated runs.

We tested these changes with all three tasks we described in the previous chapter. For reading in the midi files we used an existing library by Tuomas Eerola and Petri Toiviainen [20].

4.4.1 Slow neurons

For the slow neurons simulation, a fraction of the neurons in the liquid is randomly chosen to be slow neurons. These neurons are supposed to have a longer fading memory, but also spike less, than normal neurons. In order to accomplish this, we experimented with changing the parameters of the neurons used in the liquids.

The most important parameters for this are the membrane resistance, membrane capacitance and the firing threshold. Increasing the membrane resistance reduces the decay of the membrane potential, which can increase fading memory duration. This doesn't make the neuron spike less, which could be accomplished by increasing the firing threshold. Instead, we increased the membrane capacitance, which accomplished both of these at once, because it dampens the influence of incoming spikes on the membrane potential and also decreases the decay.

Figure 4.1 shows the result of our experimentation. The first graph shows randomly generated input spike train, the second and third show response and membrane potential of a 'normal' neuron, while the third and fourth show the response and membrane potential of a slow neuron.

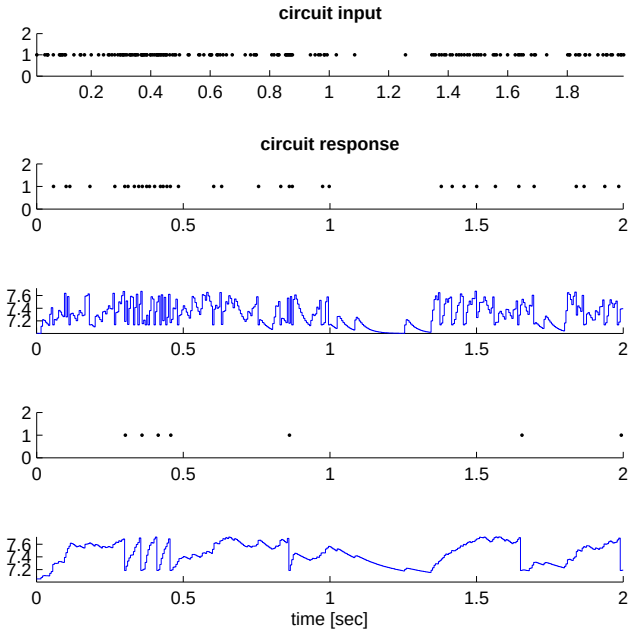


Figure 4.1: Response of a single normal neuron (2nd and 3rd plot) and a single slow neuron (4th and 5th plot) to a randomly generated spike train (1st plot).

4.4.2 Repeated runs

A method to repeat runs for statistically significant comparisons was added. The goal was to discover how different percentages of slow neurons and different hierarchical column setups would affect performance, compared to the basic task.

For every run the sample set is randomly divided into a 20% test set and an 80% training set. The task is then run using this set for all the different desired parameter settings (such as 0, 10 and 20% slow neurons). This is then repeated upwards of 100 times and the performances of every run are collected. These data can be used to make a statistically more significant comparison of the performance of different parameter settings.

Chapter 5

Experimental Results

In this chapter we will take a look at the results from both our original C♯ implementation and the Matlab Learning-Tool version. We will only discuss the actual results in this chapter, conclusions will be discussed in Chapter 6.

5.1 Using the C♯ implementation

As discussed in Section 4.2 the C♯ implementation gave no experimental results for our initial research questions. It did give us important experience with the robustness, or lack thereof, of LSMs. One thing that was particularly worrisome was the effect of high-energy signals on the activity in the liquid, often causing it to explode. This might be decreased by introducing damping effects, such as those of dynamic synapses, which we hadn't implemented yet. It would be an interesting experiment to measure at what signal energy levels the liquid still behaves as desired, with and without dynamic synapses.

The research for our own implementation also gave us insight into the choices that have to be made during the implementation of LSMs. Besides this, we also did a few experiments with distance metrics for spike trains, which we will again come back to later.

5.2 Using the Matlab Learning-Tool

The results of our experiments are shown in Figures 5.1 through 5.9. The x-axis in these plots shows the different parameters for which the experiment was

run, specifically the number of layers or fraction of slow neurons in the liquid. The y-axis measures the fraction of samples that was correctly classified in the experiment.

The results are presented as boxplots with outliers. This form was chosen because it allows us to quickly and easily compare the different results. The boxplots consist of the following parts: The centre line is the median, the edges of the box are the 25th and 75th percentile and the 'whiskers' extend to the outmost datapoints not considered outliers. Points are considered outliers when they are more than 1.5 times the difference between the 25th and 75th percentile above or below the median, this is roughly +/- 2.7σ or a coverage of 99.3 percent for normally distributed data.

In each of the following boxplots the left-most boxplot represents the results from running a task with an unmodified LSM as implemented by Maass et al. in the Learning-Tool. In the music recognition tasks, based on Papes samples, the performance on the base cases is at least as good as the results Pape shows for his implementation [18].

Slow neurons, speech recognition

Our first experiments were done with 0 to 20 percent slow neurons in a liquid and up to about 200 runs on the speech recognition task. The results from these experiments showed a very small performance difference, compared to the differences within the groups. A plot of this experiment showed very long whiskers and large numbers of outliers. This meant that any difference between the groups was likely to be insignificant compared to the differences within the groups, leading to unreliable results.

This prompted us to do a very long experiment of 500 runs (which ended up being about a week on a Pentium IV 2.8GHz with 1GB of RAM). The results from this experiment can be found in Figure 5.1. This plot shows performances between 70 and 78 percent depending on the fraction of slow neurons. There is a slight increase at 10 percent slow neurons, but it rapidly decreases after that.

The performance difference between the whiskers for each of the fractions of slow neurons is easily 40 percent, or more. Basically, the problem from our first runs remained, but because of the large number of runs, conclusions may be drawn based on these results. At any rate, it's clear that the use of slow neurons in speech recognition didn't significantly improve performance. At larger percentages of slow neurons they clearly do degrade performance.

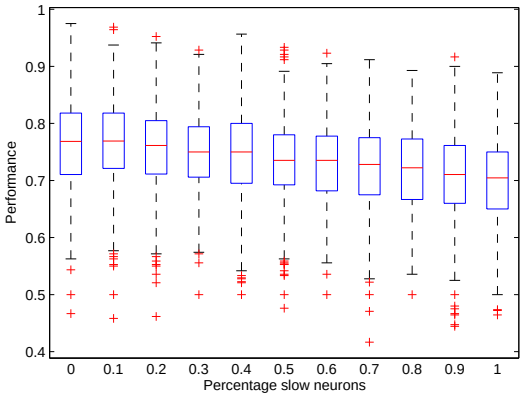


Figure 5.1: Speech recognition: 500 runs with 0 to 100 percent slow neurons

Slow neurons, music recognition

Experiments with slow neurons for music recognition showed very similar results to the speech recognition experiment, as can be seen in Figure 5.2. This experiment only goes up to 20 percent slow neurons in order to save time, because the music recognition experiment takes longer per run than speech recognition.

The 10 percent slow neurons shows a slight decrease in performance while 20 percent shows a slight increase in performance compared to a standard liquid, but these variations are probably attributable to the influence of random liquid generation, as it seems highly unlikely for 10 and 20 percent slow neurons to shift performance in different directions like this. The experiment consisted of 200 runs, less than half of that in Figure 5.1, making the results possibly less reliable than those of Section 5.2. This said, there is no reason to assume a different outcome with more runs.

Slow neurons, Maass' tasks

Applying our LSM variants to Maass' original tasks, allowed us to compare our performance with their original article. The first thing that stands out in Figure 5.3 is that the performance difference within the groups is much smaller than in the previous two plots, as can be seen in the short whiskers of the boxplots. The influence of the random liquid generation clearly has a smaller effect on these tasks, than on speech and music recognition. This is an important observation which we will certainly get back to in our conclusion. The other point of these plots is to show once more that our changes decrease LSM performance, even if

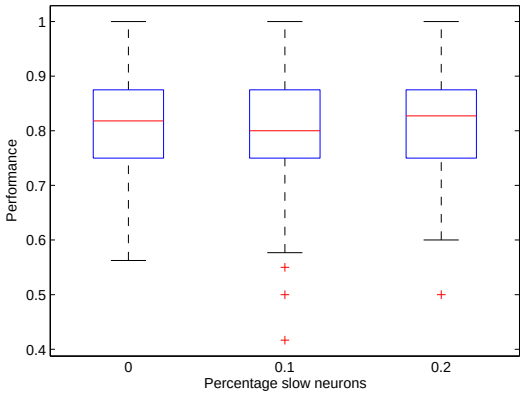


Figure 5.2: Music recognition: 200 runs with 0 to 20 percent slow neurons

only slightly.

Structures, speech recognition

This is the first of the structured liquid experiments. Each of the subfigures shows the results of 100 runs of the speech recognition task with the corresponding structural variation from Section 2.6. Unfortunately, this experiment ran a lot slower than the slow neurons experiments. This means that the number of runs for these experiments is significantly less, making the results less reliable.

Based on Maass' article we expected parallel liquids to improve performance, because this is a strict increase in computational power. The results in Figure 5.4 show differently. Although the decrease in performance is very slight, it's not entirely insignificant with 100 runs. This contradicts our expectations based on Maass' article [1].

The results of the other structures are more pronounced: Increasing the number of layers decreases performance, especially at 3 layers.

Structures, music recognition

Next we look at the result of structured liquids on the music recognition task, in Figure 5.5. Each of the subfigures shows the results of 25 runs of the music recognition task with the corresponding structural variation from Section 2.6.

Although 2 parallel layers give some improvement, 3 layers don't, at least on average. The longer whiskers show that performance with 3 liquids varies signifi-

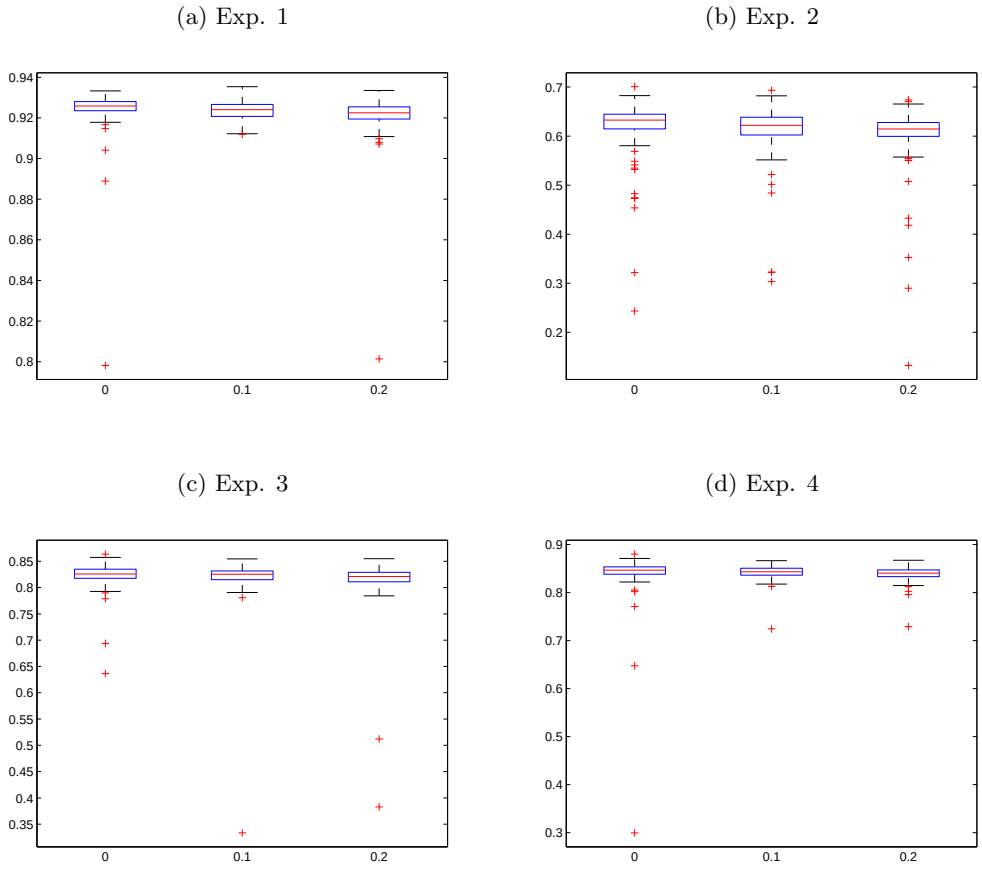


Figure 5.3: Maass' tasks: 200 runs with 0 to 20 percent slow neurons. Exp. 1: Sum of rates integrated over the past 30ms; Exp. 2: Sum of rates integrated over the past 200ms; Exp. 3: : Spike coincidence between inputs 1 and 3 in the last 75ms; Exp. 4: Spike coincidence between inputs 2 and 4 in the last 75ms

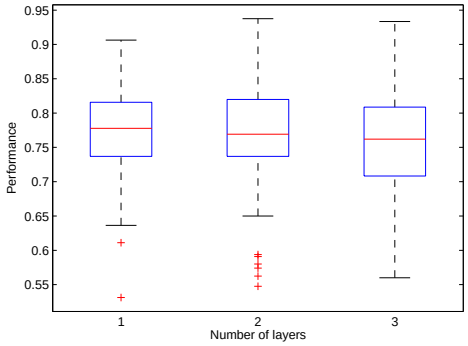
cantly more than with 1 or 2 liquids, but only for parallel liquids, but this might be statistically insignificant and disappear with more than 25 runs.

For the other structures we see only decreases in performance, from very slight at 2 layers to somewhat bigger at 3 layers. Again we conclude that these results contradict our expectations, especially regarding parallel layers.

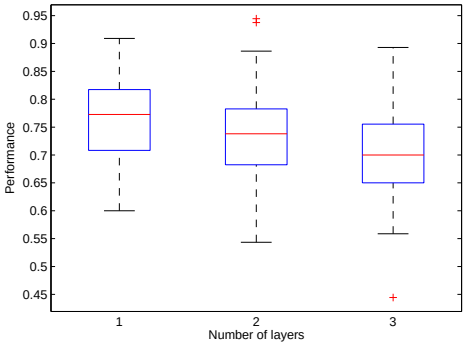
Structures, Maass' tasks

The results from the final experiment, using structured liquids for Maass' tasks, are shown in Figures 5.6 to 5.9. Although these runs are based on only 25 runs, like the experiment in Section 5.2, the results are quite different. The whiskers of

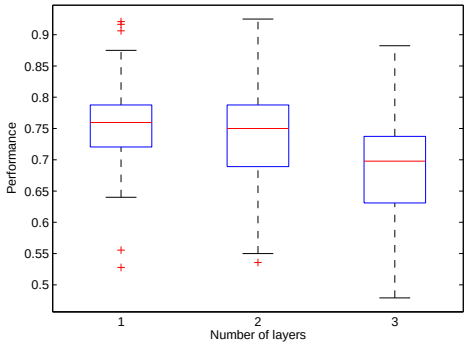
(a) Parallel liquids



(b) Serial liquids



(c) Left hierarchy



(d) Right hierarchy

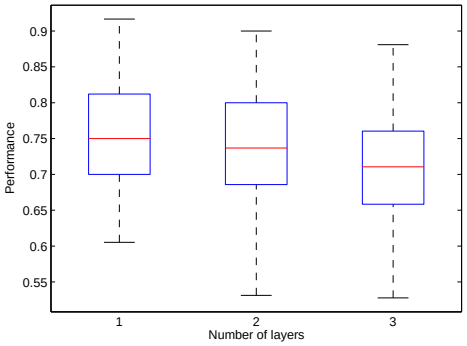
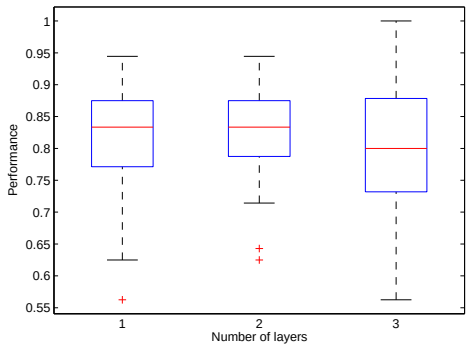


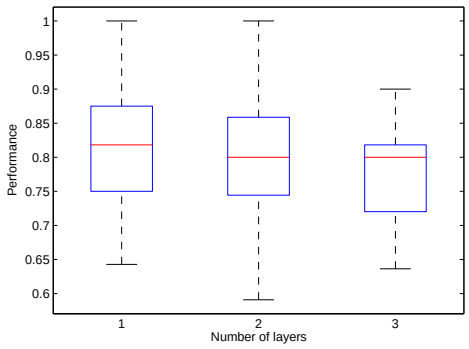
Figure 5.4: Speech recognition: 100 runs with 1 to 3 layers in all 4 structural variations

the boxplots are much shorter than those in Section 5.2, similar to those of section 5.2, although variance increases enormously with more layers. We conclude that there's a big difference between our own tasks and those from Maass' article [1].

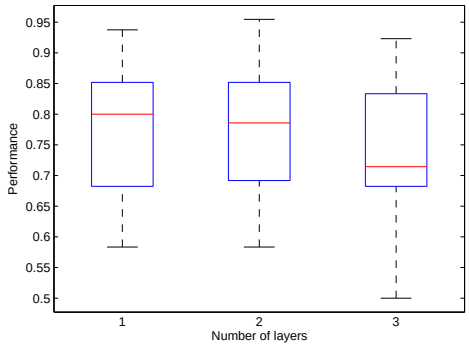
(a) Parallel liquids



(b) Serial liquids



(c) Left hierarchy



(d) Right hierarchy

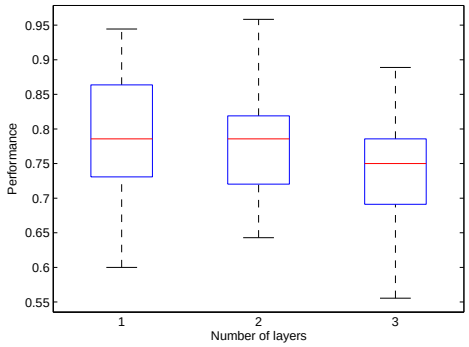
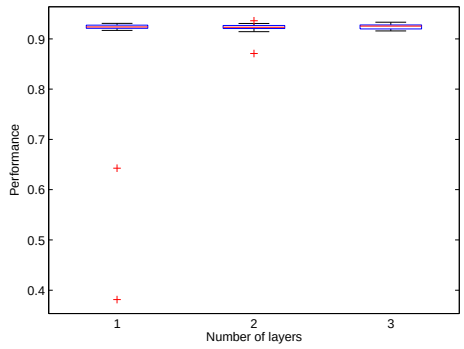
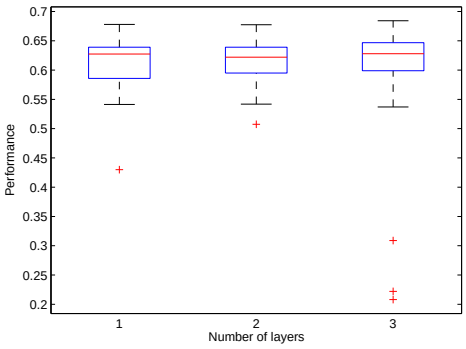


Figure 5.5: Music recognition: 25 runs with 1 to 3 layers in all 4 structural variations

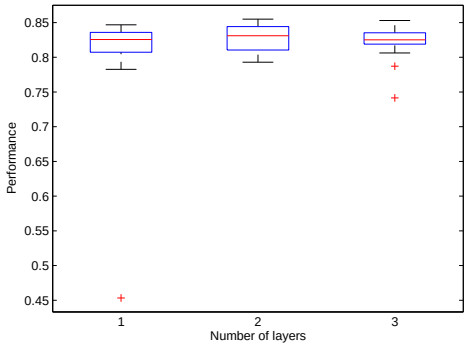
(a) Exp. 1



(b) Exp. 2



(c) Exp. 3



(d) Exp. 4

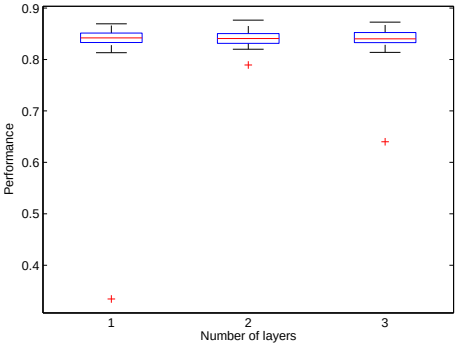
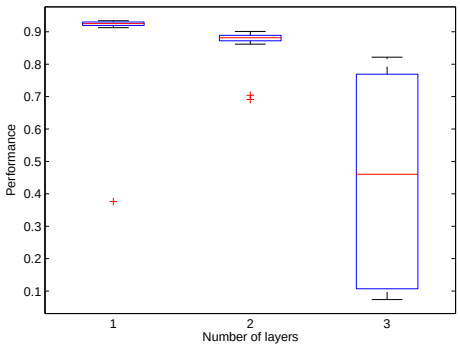
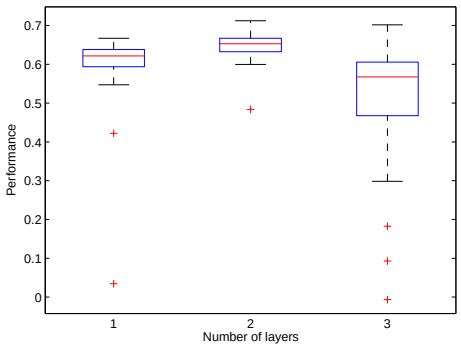


Figure 5.6: Maass' tasks: 25 runs with 1 to 3 layers in the parallel structural variation. Exp. 1: Sum of rates integrated over the past 30ms; Exp. 2: Sum of rates integrated over the past 200ms; Exp. 3: : Spike coincidence between inputs 1 and 3 in the last 75ms; Exp. 4: Spike coincidence between inputs 2 and 4 in the last 75ms

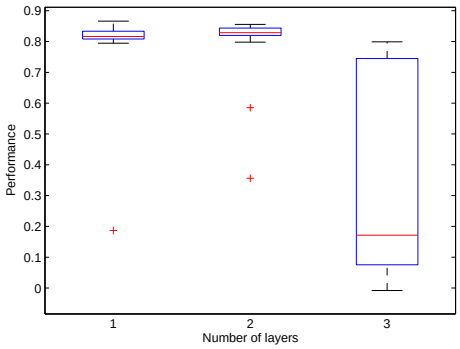
(a) Exp. 1



(b) Exp. 2



(c) Exp. 3



(d) Exp. 4

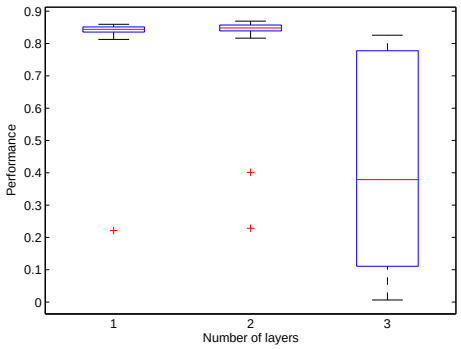
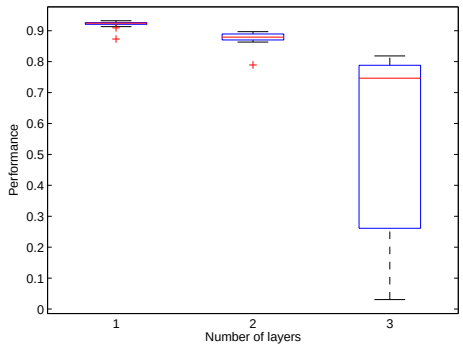
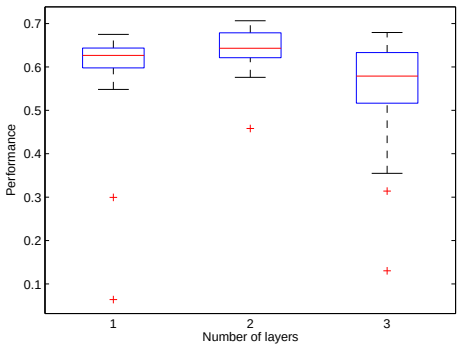


Figure 5.7: Maass' tasks: 25 runs with 1 to 3 layers in the serial structural variation. Exp. 1: Sum of rates integrated over the past 30ms; Exp. 2: Sum of rates integrated over the past 200ms; Exp. 3: : Spike coincidence between inputs 1 and 3 in the last 75ms; Exp. 4: Spike coincidence between inputs 2 and 4 in the last 75ms

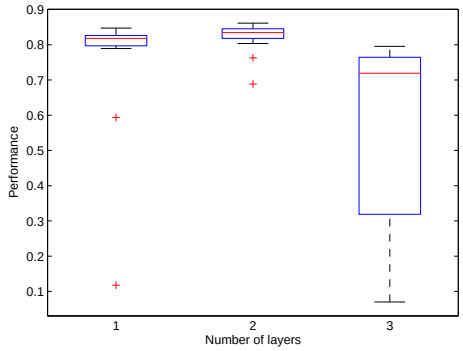
(a) Exp. 1



(b) Exp. 2



(c) Exp. 3



(d) Exp. 4

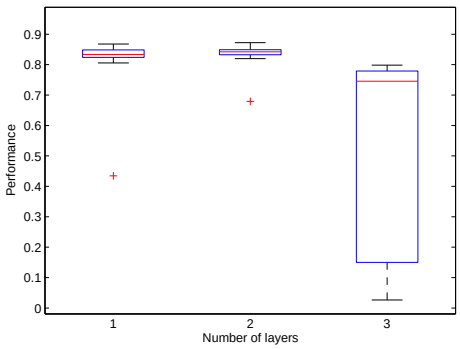
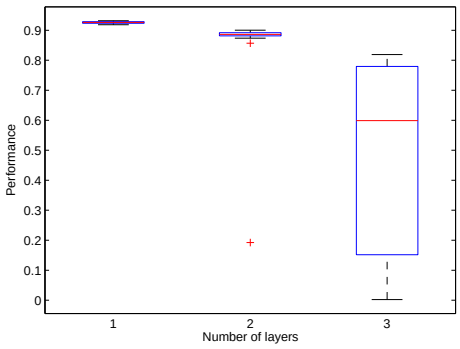
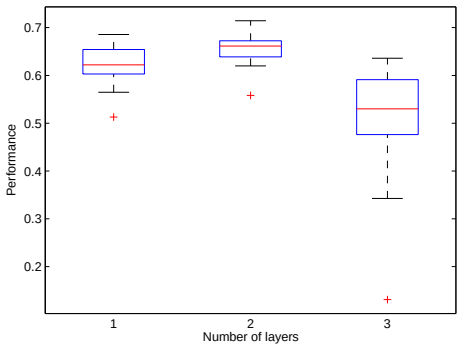


Figure 5.8: Maass' tasks: 25 runs with 1 to 3 layers in the left hierarchy structural variation. Exp. 1: Sum of rates integrated over the past 30ms; Exp. 2: Sum of rates integrated over the past 200ms; Exp. 3: : Spike coincidence between inputs 1 and 3 in the last 75ms; Exp. 4: Spike coincidence between inputs 2 and 4 in the last 75ms

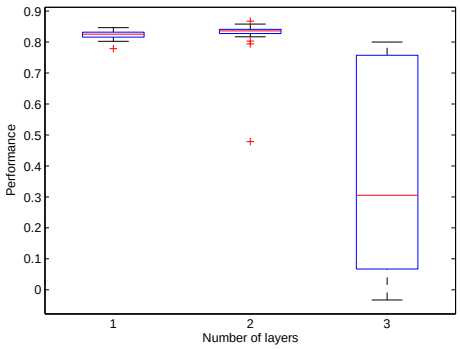
(a) Exp. 1



(b) Exp. 2



(c) Exp. 3



(d) Exp. 4

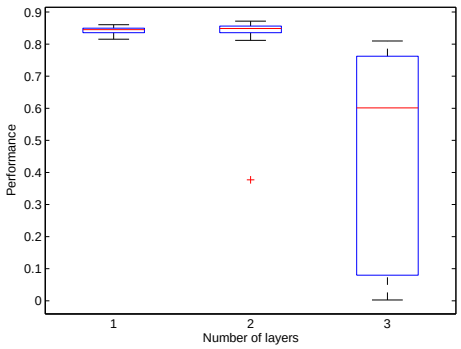


Figure 5.9: Maass' tasks: 25 runs with 1 to 3 layers in the right hierarchy structural variation. Exp. 1: Sum of rates integrated over the past 30ms; Exp. 2: Sum of rates integrated over the past 200ms; Exp. 3: : Spike coincidence between inputs 1 and 3 in the last 75ms; Exp. 4: Spike coincidence between inputs 2 and 4 in the last 75ms

Chapter 6

Conclusion

Based on the previous chapter, we conclude that none of the changes we made to the liquid, improved LSM performance to any significant degree and actually decreased performance in many cases. However, there are two important conclusions that can be drawn from our research:

Randomly generated liquids In both of our new tasks, the difference in performance between different runs is very large. The random generation of the liquids is the most important factor in LSM performance. The effects of our changes to the LSMs, were relatively insignificant compared to this factor.

This means that any future research looking to improve LSMs should probably look at replacing the random generation with something more sophisticated. In some ways this goes against the main strength of LSMs, ease of training, but the performance difference that can be achieved is of such magnitude, that it is likely to be a worthwhile pursuit.

Realistic tasks The use of abstract tasks, such as those in the article by Maass et al.[1], strongly limits the usefulness of experimental results. Real-world tasks like speech or music recognition are far better at showing the strengths and weaknesses of LSMs.

We base this conclusion on the striking lack of the performance differences observed on Maass' tasks. The effect of randomly generated liquids discussed above, is much smaller when looking at the abstract tasks of Maass et al. This can be seen by comparing any of the results from the previous chapter on Maass' tasks

to their corresponding counterpart on our tasks. For example, compare Figure 5.3 to Figure 5.1 and 5.2.

Both results for our tasks show large boxes and even larger whiskers, indicating a large spread in results. The results from Maass' tasks, on the other hand, show very small boxes and whiskers. The same thing can be seen by comparing the results of our structural variations.

6.1 Research Questions Answered

In this section we will answer the research questions from Section 1.2 in more detail.

6.1.1 Slow neurons

All of our experiments with slow neurons showed a decrease in performance with larger fractions of slow neurons. This decrease in performance is likely caused by the slow neurons' lower spiking rate. This leads to a net decrease in network activity, without creating significant new output for the readout networks to use.

The liquid's fading memory, as used by the readout, isn't positively affected by the slow neurons. Our best explanation is that the liquid's fading memory isn't found in the individual neurons' membrane potentials, but in the spikes passing between the neurons.

In our experiments the spikes from a slow neuron are treated like any other spikes by its post-synaptic neurons. Because the connections are randomly generated. Any potential extra information from these neurons' slow firing rates gets lost in the tumult of activity.

In future research, in a liquid that was trained or evolved to perform well on a certain task, the input from these slow neurons could be weighted differently. The neurons might serve their role as longer fading memory for the liquid if their spikes had a larger influence on the liquid activity.

6.1.2 Structured liquids

In the case of structured liquids we also see performance decreases, but there are a couple of things to note here. While Maass' et al. describe multiple parallel columns (see Figure 3.2) as a strict increase in computational power for LSMs, this can't be seen in our experiments.

On Maass' tasks (Figures 5.3 and 5.6 trough 5.9), there seems to be slight increase using parallel liquids, as expected. On our speech and music recognition tasks (Figures 5.5 or 5.4 for example), one can see a decrease instead, especially at 3 layers.

This means that it may not be possible to simply add computational power in the way Maass et al. propose. This is quite an unexpected result and it would be interesting to see if this holds for other tasks as well.

The other structures (serial and left- and right-facing hierarchies), show only a decrease in performance. We see a number of possible explanations:

- The readout doesn't read from the layer in which the input is injected, but a subsequent layer instead. The relation between the readout moment and the input gets even less direct, possibly making the task harder.

- The connections between the layers lead to different activity levels in the liquids than the original inputs. For example, in the serial structure, there could be as many connections between the first and second layers, as there are between the input and the first layer. However, the activity in the first layer is very different from the input activity and depending on the input could already be very high (or very low). This effect is then multiplied in the second layer, leading to exploding (or extremely low) activity. Further research may show that tweaking the connections between the layers specifically to the input of a certain task could reduce this effect.

- In hierarchical liquids, the above problem is even larger than in serial liquids. When a previous layer has more than a single column, the activity for the subsequent layer becomes much higher, simply because of the number of inputs. With every layer added, this effect is multiplied, possible leading to very unstable liquids. Again, close monitoring and tweaking of parameters specific to the task and input used could reduce this to an extent.

- The increase in the number of liquid columns simulated in the parallel and left-hierarchical cases and the resulting increase in size of the readout moments, lead to a significant increase in training time. There is also the additional hidden effect of the higher activity in the liquid allowing fewer neurons to 'sleep' (see Section 4.3). While this increase in computation time doesn't affect performance directly, it does limit the number of runs that can be done. When selecting the best LSM for a task or when simply evaluating a large number of LSMs, the net effect is another decrease in performance.

6.2 Future research

In this section we will discuss a number of possible directions for future research in improving LSMs. These are all subjects we simply didn't have the time to explore or only discovered upon studying the results of our experiments.

6.2.1 Improving liquid separation

One of the most significant conclusions of our research is that the influence of the random generation of the liquid connections far outweigh any of the changes we made, as can be seen in the boxplots in Chapter 5. This shows a clear direction for future research in LSMs. It is necessary to find a better way to create the best liquid for a task.

Stefan Kok has already done some work in the direction of improving liquid separation. Liquid separation is the degree to which the activity in the liquid is different for different inputs. Even though his results were somewhat disappointing, the idea of using evolutionary algorithms or reinforcement learning to improve LSMs is very good.

We have good hopes that combined with evolved or otherwise trained liquids, both slow neurons and hierarchical liquids would be of more use. The slow neurons could get to play a different role, where their longer term activity influences the liquid fading memory.

In hierarchical liquids, adding columns could be beneficial. It could create specialized columns for certain subtasks. For example, in speech recognition, one column might become more sensitive to vowels while another column becomes more sensitive to consonants.

6.2.2 Separation measures

The main problem Kok ran into is the computation time required to train and evaluate the performance of a complete LSM, because evolutionary algorithms and reinforcement learning both require a large number of iterations. He solved this by using a different way to measure liquid performance, by using the separation measure from Maass et al. This measure is simply the Euclidian distance between two readout-moments, calculated as:

$$S(s_k, s_l) = \sqrt{\sum_{i=1}^{n} (N_{ki} - N_{li})^2} \tag{6.1}$$

where s_k and s_l are the readout moments for two different samples and N_{ki} and N_{li} are the readouts for neuron i for sample k and neuron i for sample l. Unfortunately increasing this separation measure doesn't lead to an increase in LSM performance according to Kok.

One of the things that could have influenced this result is the simplicity of Maass' separation measure. Although it does seem like a good distance measure for the readout moments, but it apparently doesn't represent the information the readouts actually use very well. One of the places where it fails is that even the slightest shift in timing, for example delaying the input sample by a few milliseconds, can lead to a large separation according to this measure.

There are, of course, distance measurements which take this into account, such as Minimum Edit Distance, which could easily be applied to the actual liquid activity (measured as pulse trains, membrane potentials or both). For example, let's look at the liquid activity measured as pulse trains. If we have a pulse train that's simply a repetition of spikes every other millisecond, if we shift this by 1ms, it would measure as the maximum possible distance according to 6.1. Using minimum edit distance, it would be measured as a very minor distance instead.

It is interesting to note here, that there are (at least) two ways that minimum edit distance can be applied to pulse trains, which have significantly different results. One can use the pulse timings or the pulse intervals as the basis for the distance. For a study of spike train theory and algorithms we would like to refer you to [5], in which a number of different spike train distance metrics are discussed, among other things. Nortons 2008 thesis [4] also contains a discussion of separation metrics.

6.2.3 Increasing simulation speed

Another possible direction of research could be the improvement of simulation speed. If the simulation is twice as fast, we can use larger liquid columns or more complex neuron models. The use of larger liquids could already lead to some improvements, but far more interesting is the application of this speedup to Kok's proposed liquid evolution.

One method is making use of GPGPU (General-Purpose computing on Graphics Processing Units). We think this is currently the most promising method of speeding up simulations, because GPU speeds are increasing much faster than CPU speeds, at the moment. Although it is somewhat old and the source (nVidia, a GPU manufacturer) is somewhat suspect, Figure 6.1 gives an idea of current trends.

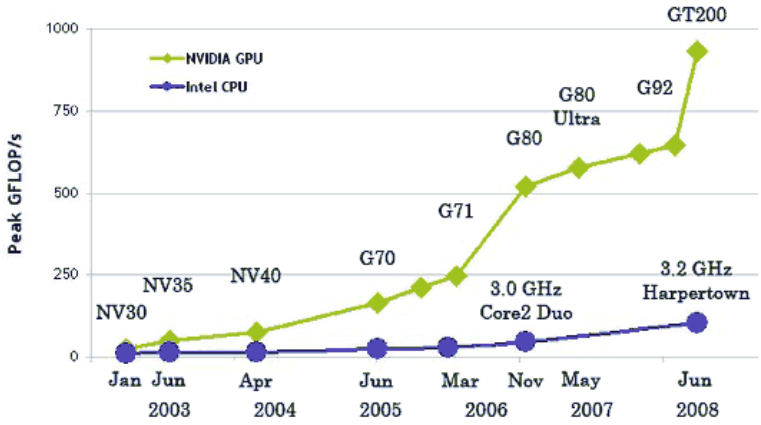


Figure 6.1: Changes in processing power for nVidia GPUs and CPUs

However, it must be noted that GPUs are really only good at a very specific type of computation, matrix multiplication. There are also quite a few constraints one needs to follow in order to create a fast GPGPU program. Specifically, GPUs are good at performing the exact same computation (down to the instruction level), at the same time on a large number of different inputs, this means that conditional code will quickly slow down GPU computations. There is also the aspect of memory alignment and other low-level considerations, but these are discussed in detail in the CUDA Programming Guide[24].

LSMs look to be easily implementable on GPUs because of how they work. Every timestep a large number of neurons is updated in the exact same way and with a little work it can likely be implemented mostly as matrix manipulations.

6.2.4 Independence of liquid and readout

An interesting experiment, which we didn't find the time to do, would be to generate a single liquid and train a large number of different readouts on it using the same task. Then taking a single randomly initiated readout and train multiple instances of it on a large number of different randomly generated liquids, with the same task. This would give us a better idea what the room is for LSM performance improvement by improving either the liquid or the readout independently of the other.

6.2.5 Improving readout approximation

The other part of LSMs that should be looked at for improvement is the readout. There are two options, either improving the readouts as they are now (using

FNNs) or looking into different techniques for readouts.

Right now it's not clear how big a role the random generation of the readout FNN plays in overall LSM performance. The first experiment should probably be taking a single liquid and training a large number of readouts on it as mentioned in Section 6.2.4. If there appears to be a lot of room for improvement of the readout, it's possible that the current training algorithm (we used Levenberg-Marquardt) gets stuck in local maxima. If on the other hand, the performance doesn't differ much between runs, it's a better idea to look at entirely different techniques, that are better at using the information present in the liquid activity.

6.2.6 Limiting randomness

In order to get a better measure of the performance differences caused by any changes, it is likely worthwhile to reduce the random influences from other parts of the LSM.

In our tasks we randomly selected fractions of the samples as training and test samples every run. Doing this only once for all runs, would remove the influence of this on the performance.

When constructing an LSM, the starting connection weights of the readout are randomly generated, as is normal in FNNs. By only creating a single set of starting weights and reusing and retraining this on different runs, one can remove another random factor from the results.

Bibliography

[1] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002. [cited at p. 1, 3, 6, 23, 26, 31, 33, 39, 48, 50, 57]

[2] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. Technical Report GMD Report 148, German National Research Center for Information Technology, 2001. [cited at p. 1]

[3] Stefan Kok. Liquid state machine optimization. Master's thesis, Utrecht University, 2007. [cited at p. 4, 6]

[4] David Norton. Improving liquid state machines through iterative refinement of the reservoir. Master's thesis, Brigham Young University, 2008. [cited at p. 4, 61]

[5] Jonathan D. Victor and Keith P. Purpura. Metric-space analysis of spike trains: theory, algorithms and application. *Network: Computation in Neural Systems*, 8(2):127–164, May 1997. [cited at p. 5, 61]

[6] D.H. Hubel and T.N. Wiesel. Receptive fields, binocular interaction, and functional architecture of cat striate cortex. *J. Physiol. (Lond.)*, 160:106–54, 1962. [cited at p. 6]

[7] Nederlandse Taalunie. Corpus gesproken nederlands. http://lands.let.kun.nl/cgn/, March 2004. [cited at p. 7, 31]

[8] Google. Google voice. https://www.google.com/voice, March 2009. [cited at p. 8]

[9] Avery L. Wang. An industrial-strength audio search algorithm. In Sayeed Choudhury and Sue Manus, editors, *ISMIR 2003, 4th Symposium Conference on Music Information Retrieval*, pages 7–13, http://www.ismir.net,

October 2003. The International Society for Music Information Retrieval, ISMIR. [cited at p. 8]

[10] L. Lapicque. Recherches quantitatives sur l'excitation lectrique des nerfs traite comme une polarisation. *J. Physiol. Pathol. Gen.*, 9:620–35, 1907. [cited at p. 13]

[11] A.L. Hodgkin and A.F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, August 1952. [cited at p. 13, 14]

[12] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. [cited at p. 14]

[13] R.B. Stein. Some models of neuronal variability. *Biophysical Journal*, 7(1):37–68, 1967. [cited at p. 20]

[14] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002. [cited at p. 20, 41]

[15] Henry Markram, A. Gupta, Asher Uziel, Y Wang, and Misha Tsodyks. Information processing with frequency-dependent synaptic connections. *Neurobiology of learning and memory*, 70:101–12, 1998. [cited at p. 22]

[16] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. pages 267–296, 1990. [cited at p. 22]

[17] Misha Tsodyks, Asher Uziel, and Henry Markram. Synchrony generation in recurrent networks with frequency-dependent synapses. *J. Neurosci*, 20:50, 2000. [cited at p. 24]

[18] Leo Pape. Neural machines for music recognition. Master's thesis, Utrecht University, 2006. [cited at p. 31, 36, 39, 46]

[19] Stanley Smith Stevens, John Volkman, and Edwin Newman. A scale for the measurement of the psychological magnitude of pitch. *Journal of the Acoustical Society of America*, 8(3):185–90, 1937. [cited at p. 36]

[20] Tuomas Eerola and Petri Toiviainen. *MIDI Toolbox: MATLAB Tools for Music Research*. University of Jyväskylä, Jyväskylä, Finland, 2004. [cited at p. 36, 42]

[21] J. Lazzaro and J. Wawrzynek. RFC-4695: RTP payload format for MIDI, 2006. [cited at p. 37]

[22] Professor David Heeger. Poisson model of spike generation. 2000. [cited at p. 41]

[23] Thomas Natschläger. Matlab learning-tool. http://www.lsm.tugraz.at/download/index.html, March 2004. [cited at p. 42]

[24] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007. [cited at p. 62]