AI4

Liquid State Mahine
Implementation and Testing

Daniel Tang Hviid
Dahvi14@student.sdu.dk

January 25, 2019

Abstract

Liquid State Machines are a non-differential Neural Network model that contains temporal information with a reservoir hidden layer, that only needs the output layer to be trained. This paper explains the motive behind implementing a Liquid State Machine, how the theory behind such a model works, goes through the implementation of said model used in this project, and shows qualitative tests of the implementation where individual neurons are investigated. Finally, this paper shows the Liquid State Machine applied to a sequence prediction problem where the model must correctly predict the next item in the sequence.

Contents

1 Introduction

In this paper we go through the motivation behind the project, as well as the theory of the Spiking Neural Network and Reservoir computing, followed by the implementation of a Liquid State Machine in a c++ environment and the tests used to ensure that the implementation works as intended including the prediction of the next item in a repeating sequence.

A Liquid State Machine is a Neural Network model that attempts to more closely resemble a biological brain than other models. It retains temporal information, and is non-differentiable. This makes the model hard to train, which is why this project only looks at training the output layer. It works on the principle that like ripples on a pond contain the information from the stones thrown into the water, spike trains propagating from the initial activation contains information that is unique to that neuron activating.

2 Motivation

In this section we look at why Artificial Neural Networks differ from biological neural networks, and why this is worth emulating and investigating. Then we look at some of the ways they can be modified to better emulate the biology of the brain

Standard neural networks are based on mathematical models. These models share little with the functions of the brain, as they are a much oversimplified version of the biological network that constitutes brainmatter, which is done in order to take advantage of differential mathematics. As biological neural networks have so far unparallelled computational powers and uses, this report is motivated by the idea that these biological networks are worth emulating in search of more powerful and general neural models.

There are several differences between biological neural networks, such as a human brain, and artifical neural networks, such as a feed forward network. These include the temporal states, the non-differential signals that the brain cells use to communicate with which are spiking voltages, activation functions that try to mimic the sodium-pumps of the brain, as well as the overlying structure of the network seeing that nature rarely conforms itself into neat layers and lines.

Looking only at the first three items on this list, there is an artifical neural network model that does take these into considerationg and as such better immitates the workings of the brain. This is the spiking neural network model, which uses the concept of spike trains instead of linear transformations to process the input information. These are said to have higher expressional power than standard neural networks, but in return require more computational power to run. The last item is structure. A biological brain is not made to be uni-directional, such as a feed-forward neural network, but instead is much better expressed as a reservoir computing framework. When one combines these two extensions of the feed-forward network, one arrives at what is called a Liquid State Machine.

3 Theory

In this section we look at the teory behind Spiking neural networks, reservoir computing, and Liquid State Machines, assuming that the reader is familiar with Artificial neural networks such as perceptrons, feed-forward networks, and recurrent networks.

3.1 Basic knowledge

The brain is a large network of individual neurons that are connected through synapses. These neurons are often dormant, waiting for the input from other neurons to fire. This pattern of a large interconnected network has produced general intelligence, and is therefore inherintly worth studying. As it is currently impossible to make a complete model, there are several concepts from biology that can be used successfully in a project such as this, including a simple Neuron model of the sodium-pump and the exitory and inhibitory synaptic connections that affect the sodium pump of other neurons through axons and dendrites.

3.2 Spiking Neural Network

The main difference between a Spiking Neural Network and a non-Spiking Neural Network is the way it processes the weighted input sum in order to find the output. For a non-Spiking NN, this is the activation function, while a Spiking NN uses a model that stores information from activation to activation. This main difference is what provides the Spiking NN with temporal information that permeates through the network.

Several such models exist, but only the Leaky Integrate-and-fire model [**?**] will be explained and used in this project.

The Leaky Integrate-and-fire model attempts to emulate the membrane potential and voltage spikes of a biological neuron. The biological neuron uses two different ion channels in order to build up potential and then dump it as a spike, which is modelled using the following equation, from [1]:

$$\tau_m \frac{dv}{dt} = -v(t) + RI(t) \tag{1}$$

Where $[\tau_m]$ is the time constant, $[v(t)]$ is the membrane potential at a given time t, R is the resistance of the membrane, and $I(t)$ is the integral of the input current. There are three different parts to this. The first is the change in membrane potential; This change in potential is the main part of the equation, as it is what we wish to find at each time step. $[v(t)]$ is the membrane potential at a given time t, which is the “leaky” part of the Leaky

Integrate-and-fire. At each timestep a part of the stored potential is lost, which is proportional in size with the stored potential. The last part is the integral of the weighted inputs, where R is the resistance of the membrane, and $I(t)$ is the integral of the input current, as with a non-Spiking NN. The time constant $[\tau_m]$ then scales all of this to fit with the time between each activation. An example of this can be seen in figure 1a.

When $v(t)$ exceeds a certain value, often set to one as an easy baseline, the model fires, producing a Dirac delta function known as a "spike" to which the model is named. This spike carries the information forward to other connected neurons as an output that can then be weighted as usual.

3.3 Reservoir computing

A reservoir Neural Network is an extension to a recurrent neural network, where the "hidden layer" is made in such a way that instead of having several layers that feed to each other in order with recurrent connections, the individual neurons are simply randomly connected to each other with random weights. This produces what is known as a reservoir that can then be connected to a readout layer that takes the place of the output layer. The two common techniques for this are known as the Echo State Network and the Liquid State Machine [1].

4 Implementation

In this section we look into how the Liquid State Machine was implemented in a code environment, and which choices were made along the way. The code implementations of this project used C++ as a language, and generally used Object Oriented design philosophy, aiming to be modular and easy to use.

4.1 Neuron

The base of the implementation is that of the neuron. This class is responsible for the heavy lifting of the implementation, as it handles the calculations of it's own state, as well as the input and output between individual neurons. Below are subsections explaining the important parameters and structures of the class, as well as a class that inherites from the Neuron class.

4.1.1 Input and outout

For every neural network, the neurons must have inputs $[i]$ and outputs $[o]$. These inputs are, with the exception of the input layer the outputs of previous neurons. Often, the input and outputs are stored in matrices, or tensors, as the inputs are calculated by applying the input weights to the activation function, which produces outputs. In this implementation, however, whenever a neuron activates it calls every neuron that it is connected to and adds the corrossponding weight to the collected input of that neuron. This is done because in a spiking neural network the output is either one or zero, and at any given activation step it is to be expected that the majority of neurons are not activating, thus saving computations. It should be noted that these savings could also be achieved through a sparse matrix or tensor, and that libraries exist that do these kinds of calculations very efficiently, but this method was chosen for simplicity of implementation.

4.1.2 Membrane potential

In non-spiking Neural networks, the input is reset for after each activation of the Neural Network; However, when using spiking neurons the input spills over to the next activation. This spillover is called the membrane potential [1], [u]. This internal value is what determines whether or not the neuron spikes to produce an output. Because a leaky-integrate-and-fire **??** model is used this value will decrease over time if it is not increased by other neurons spiking.

4.1.3 Readout neuron

In a Liquid State Machine the readout layer is a bit special. Because of the nature of the spiking models, it can be ver hard to train spiking neurons, and as such we want to do it as sparsely as possible. Because of this, having the readout layer be the only part that is trained is an enticing option. This, however, is hard to achieve if the synapses are used for readout, as it is in the standard Neuron class of this implementation. Because of this, the readout neurons directly access outputs from it's inputs. This input will then be added to the membrane potential of the neuron, bringing it closer to it's firing threshold.

4.2 Algorithm

Now that the elements are accounted for, the algorithm that is run each time the LSM is activated can be explained. As can be seen in algorithm 1, each time the network is activated it goes through four steps: **Inputs**, **Check Activity**, **Activate Neurons**, and **Readout Neurons**. **Inputs** translates the network input into inputs for the input layer, and checks if they should be added to the list of spiking neurons. Then **Check Activity** adds all active neurons to the list of active neurons, before **Activate Neurons** activates all neurons with a high enough membrane potential. This sequence ensures that every neuron calculates it's internal state immidiately after every spiking neuron fires, ensuring that every neuron accesses the their inputs from the previous timesteps. Finally, **Readout Neurons** activates it's synapses and checks if it spikes. The readout neurons needs to do this in reverse order because they pull the activations, instead of pushing them.

```
Data: Input values
Result: Readout values
Step 1 : Inputs;
forall Input Neurons do
    Apply provided input value;
    if MembranePotential ≥ SpikingPotential then
        Add neuron to Active Neurons List;
    end
end
Step 2 : Check Activity;
forall reservoir Neurons do
    if MembranePotential ≥ SpikingPotential then
        Add neuron to Active Neurons List;
    end
end
Step 3 : Activate Neurons;
forall Neurons in Active Neurons List do
    Activate all synapses;
end
Step 4 : Readout Neurons;
forall Readout Neurons do
    Activate all synapses;
    if MembranePotential ≥ SpikingPotential then
        Set output to 1;
    end
end
Output active Neurons;
```

Algorithm 1: The main algorithm of the LSM implementation. Input and readout layers are partly seperated from the reservoir for simplicity and modularity.

Finally, this leaves a set of readout values corresponding to which of the readout neurons have been activated. This, however, does not necessarily translate very well into a readable output, as readout neurons cannot be expected to spike every activation.

One way of translating the output is to look at the membrane potential of the readout layer instead of the spikes. This provides information about how close the neuron is to outputting, which is useful for training, but has the disadvantage of being low right after a spike has occured. Another approach is to average the spikes of several sequential activations, thereby looking at the frequency of the spikes. This can also be used for training, but contains less information if only a single, or few, spikes occur. This can also be done on the entirety of the reservoir as well, to get a spiking rate of the population.

Finally, one can use both. By looking at both the internal value as well as the spikes, and averaging this value over several activations, enough information is stored about the frequency of the spikes, as well as the lead-up to each spike. As the spike values should be higher than the internal value, this value should be weighted higher than one, which is the max value of the internal value. Because of this, the output should be normalized, by dividing it with the weight of the spikes.

5 Testing

This section documents the tests used to ensure that the implementation of Liquid State Machine is functional, and to highlight certain properties of both the implementation and the network model itself, finishing off with a test problem.

5.1 Neuron model

The neuron model is a critical part of this project, and as such should be tested to gain insight of what impacts the firing rate and sensitivity of the model.

In order to test this model, a baseline set of parameters must first be selected. The baseline parameters are as follows; Resistance : 2.2, Time constant : 10, Resting potential : 0, Spiking potential : 1. For these parameters, the model behaves as can be seen in figure 1a which is clearly a smooth membrane potential curve followed by a spike as the potential reaches the spiking threshold. In figure 1b, the parameters have been changed to allow a random input between zero and one. This input range is chosen to simulate a set of different synapses providing input to the neuron could behave if the expected sum of their synaptic inputs was in this range, illustrating how a neuron would act when it's input is affected by several synapses that fire independent of each other. It is clear from these two images, as the neuron fires in exactly the same timestep, that the mean input is relevant, even if the input is not consistent.

If one instead changes the parameters so that the resistance is twice of what it was in the baseline, and sets the input to a random number between zero and one each timestep, the result, as seen in figure 1c, the neuron fires three times in the same timestep, showing how the resistance is not proportional to the fire rate of the neuron.

5.2 Propagation

In a functional Liquid State Machine, the activation of a neuron in the hidden layer will spread to nearby neurons like ripples in a pond. In order to test this, one can activate a single neuron and observe this behavior through the activation of the connected neurons. One way of performing this observation

(a) The neuron model using the baseline parameters and constant input of 0.5.

(b) The neuron model using the baseline parameters and random input with a mean of 0.5.

(c) The neuron model using double resistance, and random input with a mean of 0.5.

Figure 1: Tests of the neuron model.(a) shows the ideal potential curve followed by a spike. (b) shows how the potential curve caused by a randomized input can be used in place of the ideal curve to get the same frequency of the spike. (c) shows how resistance parameter is nonproportional to the firing rate.

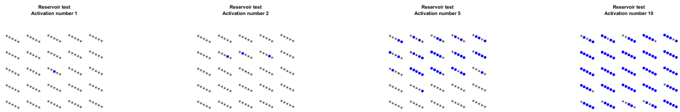
is by setting the neuron parameters in such a way that a single received pulse is enough to cause an activation, and then activating a single neuron in the reservoir. The expected behavior is a cascade effect where each activation will activate several other neurons, eventually activating each connected neuron in the reservoir.

By visualizing the activity of the network through repeated activations, this is easy to test. In figure 2, it is clear that a single reservoir neuron is activated after the first timestep, and that after ten timesteps all connected neurons have been activated. In this figure it is also clear that three adjacent neurons, which is the number of connections each neuron have, are activated in the second timestep, as is expected from a functional Liquid State Machine.

5.3 Reservoir activity

The reservoir parameters are as important as the neuron parameters. There are several parameters that can be tuned and modified, and in this section the number of connections as well as the proportion of exitory to inhibitory neurons are tuned. The rest of the parameters are kept constant, as well as the neuron parameters, which are as follows; Resistance : 2.2, Time constant : 10, Resting potential : 0, Spiking potential : 1, Minimum weight : 0, Maximum weight : 1, Max synapse length : 2, Number of input neurons : 3, Number of connections per input neuron : 10.

Having too few connections will result in low amount of activity that won't spread beyond the input neurons. Conversely, having too many connections will result in a cascade of activity that results in all, or most, neurons activating every timestep. Both of these options are equally useless, as they provide essentially constant output.



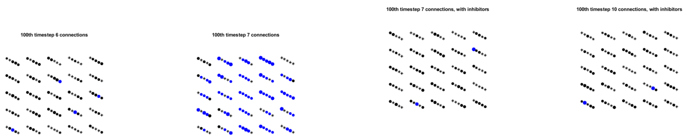
(a) The test network after a single timestep.

(b) The test network after two timesteps.

(c) The test network after five timesteps.

(d) The test network after ten timesteps.

Figure 2: Test of the propagation of the network. Every neuron has three synapses which are adjacent to it, and each spike will activate all three synases. After the first timestep, only the selected neuron is active. In the second timetep, it has spiked and is therefore inactive while the three neurons it has synapses to are active. After five timesteps a large proportion of the neurons are active, and after ten all connected neurons are active.



(a) The test network after a hundred timesteps with 6 connections per neuron.

(b) The test network after a hundred timesteps with 7 connections per neuron.

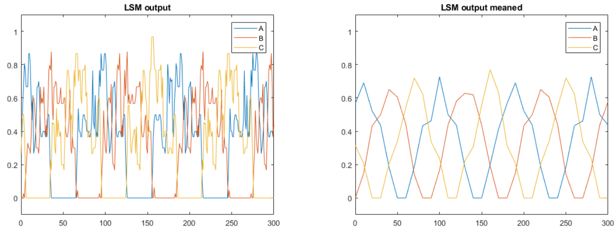
(c) The test network after a hundred timesteps with 7 connections per neuron, with inhibitors.

(d) The test network after a hundred timesteps with 10 connections per neuron, with inhibitors.

Figure 3: Spikes and membrane potential after a hundred timesteps with different amounts of connections per neuron and inhibitors.

First, to test the impact of the number of connections that each neuron have on the level of activity of the reservoir, two different amounts of neuron connections are tested. The reservoir after one hundred timestept with six and seven connections per neuron can be seen in figure 3a and 3b. As can be seen in the first image, six connections are enough to provide a good level of activity throughout the network, without having a lot of the neurons activate each timestep. On the other hand, having seven connections is enough to cause a run-away reaction that activates almost all the neurons each timestep. Clearly, there is a fine balance between having enough connections, and having too many.

A way of correcting this is by including inhibitory neurons. In figure 3c we see how 15% inhibitory neurons brought the level of activity back down, and in figure 3d.



(a) The last 300 timesteps for the test data.

(b) The last 300 timesteps for the test data, meaned outputs.

Figure 4: Outputs for the sequence prediction test.

5.4 Sequence prediction

In order to test the full implementation, a test problem was chosen. This test problem is a classification problem on a continous sequence of "AAABB-BCCC", using the Liquid State Machine to predict the next input in the sequence.

In order to train the network to this sequence, every input is continued for ten timesteps, equal to the time constant. The parameters for the neurons and reservoir are; Resistance : 3.3, Time constant : 10, Resting potential : -0.1, Spiking potential : 1, Minimum weight : 0.5, Maximum weight : 1, Max synapse length : 1, Number of input neurons : 3, Number of connections per input neuron : 100, Learning rate : 0.0001, Proportion of inhibitory neurons : 25%.

The training and testing is carried out by first creating the input for 1.200.000 timesteps. The first 90% is used for training, and the remaining 10% for testing.

In image 4a the outputs for the last 300 timesteps are shown, and in image 4b the outputs for the ten timesteps per sequence element is meaned. It is clear from the first image that the signal is somewhat unstable, which is supported by the accuracy of the network when looking at individual timesteps, which is 82%. The second image, in contrast, is much more stable, and indeed has an accuracy of 100% for the entire test set.

5.5 Discussion

In this section we discuss the Liquid State Machine, more specifically the choises taken when implementing it, the troubles of tuning it, and the results of the tests.

When implementing a neural network, one would usually make a set of matrices and tensors that contain the inputs, outputs, activations and weights.

In this implementation, however, a graph implementation with nodes for neurons was used. This impacted the computational time of the network, but also allowed for easier bugfixing and faster implementation. This tradeoff was made because of the limited scope of the project, as well as the relatively small problem it was used on. This allowed for the implementation of the Neuron class, which made it much easier to implement it in a modular fashion.

Tuning a Liquid State Machine is a tedious process, as a small change in parameters will make the network lose a lot of accuracy. In particular the number of connections, neuron resistance, resting potential, proportion of inhibatory neurons, and number of synapses per input neurons. These parameters should be tuned quite gently, as they will make the network either almost inactive or start a chain reaction.

5.6 Conclusion

Making an artificial neural network that more closely resemble the human brain than perceptrons and similar models require spiking models as well as more reservoir-like layers. The Liquid State Machine fullfills this, as it is reservoir computing that uses spiking neuron models. This project implemented this model in a c++ environment, keeping the implementation modular to allow for easy testing. The implementation was then tested on a relatively easy case, where the Liquid State Machine was used to predict the next letter in a repeating sequence, with an accuracy of 100%. As such, the goal of implementing and testing the Liquid State Machine was achieved.

Bibliography

[1] Hélène Paugam-Moisy and Sander Bohte
Computing with Spiking Neuron Networks
pdfs.semanticscholar.org/8997/95d8e3c07976f62149eb79612e06231d6aee.pdf