Liquid State Machine Optimization

Stefan Kok

Utrecht University

December 17, 2007

Supervisors:
Dr. Marco A. Wiering
Drs. Leo Pape
Dr. Ignace T.C. Hooge

Abstract

In this thesis several possibilities are investigated for improving the performance of Liquid State Machines. A Liquid State Machine is a relatively new system that is a Machine Learning system, which is capable of coping with temporal dependencies. Basic Recurrent Neural Networks often have problems with this. One reason for this is that it takes a long time to train the Recurrent Neural Network. Liquid State Machines train much faster by using a temporal reservoir to map temporal input into a static output pattern. These output patterns can be learned by a statistical learning method. In this thesis, two different subjects are addressed. The first subject is about reducing computation time on calculating the performance. Optimization algorithms for neural networks are often computationally heavy. This is because the performance of the system, here the Liquid State Machine, needs to be evaluated. The computation time can be decreased by using other methods to evaluate the performance. The other subject that is addressed here, is in the optimization of the temporal reservoir in the Liquid State Machine. Two different algorithms are used here, namely Reinforcement Learning and a Genetic Algorithm. The goal is to find out if the algorithms can improve the performance by improving the temporal reservoir and if so, if the performance is increased that much so that it is computationally beneficial to use. The experiments using a different performance measure showed that it will probably not help in improving the performance on classification of the Liquid State Machine. The other experiment using the two different algorithms showed that Reinforcement Learning can not find a better setting for the temporal reservoir to improve the performance given the settings of the experiments. But the Genetic Algorithm is able to improve the temporal reservoir and thus improve the performance, this was tested on two different datasets. The first dataset used was a movement classification task. The results showed an improvement, but comparing it to another system, namely Evolino, the Liquid State Machine is outperformed on both classification and computation time. The second dataset used is a music classification task. The results here where more in favor of the Liquid State Machine, although it is unclear if the parameter setting for Evolino is optimal.

Contents

Chapter 1

Introduction

1.1 Introduction to Artificial Intelligence

Artificial Intelligence (AI) is a broad subject. One purpose for AI is related to explain phenomena. In brain studies for example, neurons can be simulated to explain certain types of brain patterns. But probably most of the research in AI is done on finding solutions to solve problems. This can be all sorts of problems like ordering some data or classifying objects. For most problems there is not one single method to tackle it. Often there are different approaches to solve problems in AI. For example some approaches use learning algorithms to find a solution, while others use a pre-programmed set of rules to solve the problem. The learning algorithms fall under a section in AI called Machine Learning [13]. A number of well known learning algorithms that fall under Machine Learning are Genetic Algorithms [4], Reinforcement learning [25] and Neural Networks [13]. These algorithms have an overlap in the problems they solve, but solve these in a different way. A problem with learning algorithms is that they are not always transparent. For example, an artificial neural network basically can be seen as a vector of values. This makes it difficult to explain why it behaves like it does and what these values actually mean. If the neural network does not learn like it should, it is difficult to find the problem.

1.2 Time Series and Time-dependencies

One type of problem that gets a lot of attention in AI is a problem with a temporal aspect. This means that in the input data there is a time-dependency. In other words, an input at one timestep does not contain enough information to get the correct output. The output is thus dependent on a number of input patterns. An example of a temporal problem is speech-recognition. Here the task is to classify spoken words. In this case classification means that the output is a written version of the spoken word. The temporal aspect forms a problem for Machine Learning algorithms because the temporal-dependency is often difficult to learn with the learning algorithms that are often used. The system thus needs some sort of method to store previous input patterns. For example the words 'bike' and 'like' sound the same except for one letter. If the algorithm only classifies using the input pattern at one timestep, when the letter 'i' is spoken, the algorithm will not be able to differentiate between the two words. Feed-Forward Neural Networks have trouble in dealing with this time-dependency. Because of the different lengths of samples it is difficult for these Neural Networks to have a complete memory of the input. But this is not the only problem, it is often not known in advance what the length of a time-dependency in a sample is. This makes it hard to set the number of inputs for the network to the right size.

Hidden-Markov models [19] are statistical methods that are used as an application for problems that have a time-dependency. A Hidden-Markov model consists of a set of states. The transitions of one state to the other are decided by a probability distribution in that state. The probability distributions for each state is calculated by using a large set of examples and calculating the chance to go another state. Another application that is applied to problems with a time-dependency is the Time-Delay Neural Network [11]. This can be seen as a normal Feed-Forward Neural Network (see 2.1), but instead of having input from one timestep, the input is stored for a number of timesteps and given to the network at once.

Another system based on Feed-Forward Neural Networks, is called a Recurrent Neural Network [5]. This system uses recurrent circuits in the network to cope with the time-dependency. The activation is thus kept in the network by sending the output of a neuron back into the network. Recurrent Neural Networks can be found in many forms. One is the continuous Recurrent Neural Network, where neurons use a continuous function to process the input. Another system is based on biological neurons, which is called a

Spiking Neural Network [26]. Here the neuron uses a threshold function over the input, to decide what the output should be. Spiking neurons are different from neurons with a continuous function, because they only send out two types of signals, namely a zero for not spiking, or a one for spiking. Unlike in continuous Recurrent Neural Networks, the information is just partially in the signal itself. Most of the information comes from the timing of the spike signals. If only one signal comes into a neuron, it will not fire, but if a lot of signals come into a neuron together, the probability to spike increases. Another difference with continuous Recurrent Neural Networks is that the activation in the spiking neurons is partially kept for future timesteps, which causes each neuron to have an internal memory.

A last example of a Recurrent Neural Network that is used for time-dependencies, is called Long Short-Term Memory [8]. This network is based on a continuous Recurrent Neural Network, which uses special kinds of neurons called memory cells. These memory cells also have an internal activation like spiking neurons. The memory cells are special, because they use gates. One is to block incoming activation from coming into the cell, another is to let activation within the cell leak away and the last one is to let activation from within the cell out into the network. The continuous Recurrent Neural Networks use some form of a Gradient Descent algorithm [13] to train the weights. Spiking Neural Networks are different. Although there is a Gradient Descent algorithm, which is called Spike-Prop [3], there are other forms of learning. One is called Hebbian Learning [6]. This is a learning algorithm uses some simple rules to change the weight between two neurons. If the two neurons spike at the same time, the weight will increase, if only one spikes, the weight will decrease. All the neural networks described above have problems to deal with time-dependencies, the Recurrent Neural Networks all have trouble with learning the time-dependencies and all the neural networks suffer from the problem that it takes long to optimize the weights.

Two relatively new systems that cope with time-dependencies in a different way are Echo State Networks [10] and Liquid State Machines [14]. These two systems are based on the same idea. The two systems use a Recurrent Neural Network to create static patterns of the temporal data. These static patterns are then learned by a statistical learning algorithm. The difference between these two systems is that in Echo State Networks, continuous Recurrent Neural Networks are used, while in Liquid State Machines, Spiking Neural Networks are used. The advantage is that both systems get round the troubles of learning in the Recurrent Neural Network, instead learning is done using a relatively simple algorithm which does not take that much time to train.

1.3 Research Question

Liquid State Machines (LSM) can be computationally demanding. This forms a problem when improving the LSM. For example to improve the liquid of the LSM, the performance of the LSM is needed which costs time. To decrease the computation time, other methods to evaluate the performance of a liquid should be investigated. One idea that could be used is a measure for the separation of a liquid. The separation defines how well a liquid can divide output patterns of a liquid given the input patterns. Using the separation property may decrease the computation time as there is no readout network that needs to be trained, but also it could be that to calculate the separation, less samples need to be used. One research question is thus:

> *Is the separation of a liquid a good and fast performance measure for liquid optimization?*

One aspect of this question is how the separation should be represented.

The second research question in this thesis is about the different algorithms to optimize the liquid. There are a number of methods to optimize the performance of a liquid, it will give a view on which algorithms are capable of optimizing the liquids and how much they will improve the performance. This part of the research may show how complex the search-space is in finding the optimal solutions. Another comparison to make is to see whether these algorithms outperform already good performing systems that can also handle time-series problems, like Evolino [23].

> *How well are several different algorithms performing in optimizing a liquid in a Liquid State Machine?*

For testing these research questions, it may be insightful to use a *toy problem* where the environment can be better controlled. A *toy problem* is a problem which is artificially created. A *toy problem* can give a fair comparison between the different algorithms and performance measurements that are used. In the experiments the implementation of this *toy problem* is a movement classification task. But an algorithm should not only be capable of handling simple *toy problems*, but also more complex real-world problems. This could show how dynamic these systems are, as they show their capabilities to handle noise and inconsistencies. Another part that is important for AI algorithms is how well they perform compared to humans. Real-world problems often already provide insight in how well humans can cope with the problem. One of these is music classification. This task is something that humans are capable of solving.

1.4 Relevance to Artificial Intelligence

Cognitive Artificial Intelligence (CAI) is about creating new systems that have intelligent behavior and study human behavior mechanisms to find new systems that are capable of performing new tasks. In this thesis the aim is at the first part, creating new intelligent systems or at least improving on them. All the systems that are inspected here, are learning systems. Learning is an important part for humans as it gives us the experience to cope with the uncertainties in life. Problems that humans have to solve are often problems with a time-dependency. For example, we are able to communicate with each other, by behavior (for example getting a red face if we feel ashamed), or with sign language or by using sound (crying, but of course also speech). In AI this is an aspect where there is a lot of research done. This is because it is difficult to find a system that can easily and with low computation time, cope with time-dependencies. Liquid State Machines are such a system that are able to deal with time-dependencies using less computation time than other known systems. The goal is to find out what improvements can be made to improve the performance of Liquid State Machines.

1.5 Outline

This thesis concerns Machine Learning systems that can handle time-series data. In chapter 2 an introduction will be given to the algorithms that will be used in the experiments. The first algorithms are Feed-Forward Neural Networks, which are used as readout networks in Liquid State Machines. Next there is an introduction to Recurrent Neural Networks (RNN) and time-series data. After this Long Short-Term Memory and Liquid State Machines are described. These two systems are then used in chapter 3 for optimization.

In chapter 3 the training paradigms that are used to optimize RNNs are further explained. These training paradigms are used to optimize the recurrent network of the Liquid State Machine and the Long Short-Term memory in Evolino. In the first section two different performance measures are explained, namely the separation of the liquid and performance of the readout network. These measures are used to evaluate the performance of a RNN for the learning algorithms. There are two algorithms that are used to optimize the RNN in the LSM. First a Reinforcement Learning algorithm will be introduced and the second training paradigm for LSMs is a Genetic Algorithm. The last section is about a system called Evolino, this combines a Long Short-Term Memory network with a regression method to learn time-series data.

In chapter 4 the datasets that are used in the experiments, are explained. There are two types of sets used. First there is the movement classification set, which is a *toy problem*. Here the algorithms need to classify in which direction a target moves on a grid. The other set is about music classification. Here the task is to classify which composer has written a certain musical piece.

Chapter 5 is about the experiments. The first section will explain which parameters are used. The second section is about the experiments that are done on the different learning algorithms. These are about the performance of the different training paradigms like Evolino and the Genetic Algorithm. But there are also a number of experiments done on the performance measurements.

In the last chapter (chapter 6), the results of the experiments in chapter 5 are discussed. Furthermore some work that can be done in the future will be discussed.

Chapter 2

Reservoir Computing

Solving time-series problems is a major challenge in Machine Learning. Time-series problems are problems that have a temporal aspect, which means that information of previous timesteps needs to be stored to be able to recognize a sample correctly. An example of this time-dependency is in speech-recognition. This gives a problem when for example a spoken word needs to be recognized. Here there should be a memory to remember the complete word when making a decision about which word it is. If the input at timestep $t-1$ is not used for input at timestep t the network will probably never find the correct word. The problem with time-series is that neural networks that are used, not only need a long-term memory, as also used in non-temporal problems, but also a short-term memory. This is because otherwise, input at a timestep is independent from the other timesteps. The problem with Feed-Forward Neural Networks is that time-series are not always of the same size. For example with speech-recognition, words are not of the same size, but a neural network has a static number of inputs, thus making it hard to define the size of the input layer. Liquid State Machines are able to solve this by using a fading memory which is then used as input for a readout network. This fading memory is a mapping of temporal data into a static representation, which should be easier to learn for the readout network.

In this chapter the systems that will be used in the chapter about the training paradigms (see chapter 3) will be explained. First Feed-Forward Neural Networks (FNN) are explained. FNNs are often used in solving problems in Artificial Intelligence. Also it gives more insight in Recurrent Neural Networks (RNN) as a lot of these networks are based on FFNs. These RNNs are explained in section 2.2. Not only will the system be discussed, but also some of its problems. Furthermore this section discusses what time-series data exactly is. After this a RNN called Long-Short Term Memory is discussed (LSTM; see section 2.3). LSTM will be further used in section 3.4. Finally Liquid State Machines will be further explained (see section 2.4).

2.1 Feed-Forward Neural Networks

A Feed-Forward Neural Network (FNN; [13]) is a statistical learning method based on some basic principles of biological neurons. Although FNN neurons do not look like biological neurons at all, they share an important characteristic that make FNNs powerful, namely parallel processing. As can be seen in figure 2.1, an FNN has a number of layers. The earliest FNNs only had

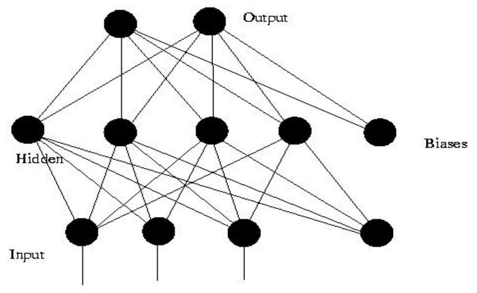


Figure 2.1: Image of an FNN with three layers, from bottom to top an input layer, hidden layer and output layer. The biases are on the right

two layers [21], the input layer and output layer. To add more computation power, a hidden layer is used, which means that it can approximate a lot more functions than without. Function approximation is used, because in most datasets, the function that can output correctly given the input patterns is not known. Function approximation does what the words say, it approximates the function that can deal with the dataset correctly. In other words, the approximation function tries to get as close as possible to the real function that creates the output. The network is built up of nodes (also called neurons), these nodes are connected through weights (real number values) and these connections are directional. In other words, the input layer provides input for the hidden layer and the hidden layer provides input for the output layer. Also both the input layer and the hidden layer have a bias node, which is an input node with a constant value namely one. This is, because otherwise every function has output zero if the inputs are zero, which decreases the number of functions it can approximate. The input for the hidden layer is calculated by multiplying the weight of the connection between the two nodes with the output of the input node. After this, a function is applied using the summed input for the node. This can be the identity function (which means nothing will be changed), but this decreases the solution space to only linear functions. A well known function that is used in FNNs, is called the sigmoid function ($\sigma(x)$, see equation 2.1). This function squashes the input so that the outcome of the sigmoid function lies between a minimum and maximum. The function can be seen in figure 2.2. The sigmoid in figure 2.2 is called a logistic sigmoid function. Here the min-

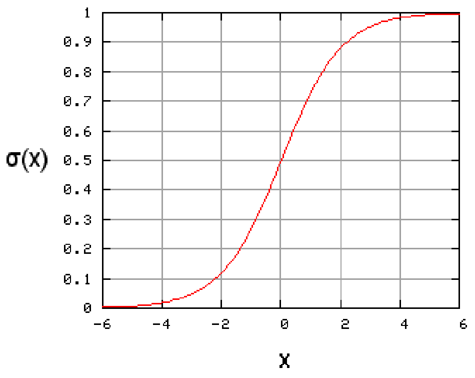


Figure 2.2: Graph of a Sigmoid function

imum outcome is zero and the maximum outcome is one. There are other sorts of sigmoids like the hyperbolic tangent function, which has its minimum at minus one instead of zero. The logistic sigmoid function is calculated as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.1}$$

The input x for $\sigma(x)$ for neuron j is calculated as follows:

$$x_j = \sum_{i=1}^{n} x_i w_{ji}, \tag{2.2}$$

where w_{ji} is the weight between node i and node j and x_i is the output of node i. This is done for both the hidden layer and output layer (and other hidden layers if there are any). But often, the transfer function of the output layer is the identity function instead of a sigmoid function. There are a variety of problems that can be solved using FNNs. One type of problems are regression problems, where the function needs to predict the output at that moment. For example with the stock exchange, it would predict the value of a certain stock. But there are also other problems that can be solved, for example classification problems. Here the output layer has a number of different output neurons, one for each class. The index of the output neuron with the highest output value will then be the class that the FNN predicts.

To find the right outputs for the input patterns, given the target values, the weights need to be adapted to find the closest approximation to the target values. One possible method to search for the optimal weights is using an algorithm called Gradient Descent [13]. Here the derivative of the network with respect to the error is taken. After this, using the derivative, a step to the steepest decline (or incline) in the search-space is taken to search for a better solution. A well known implementation of the Gradient Descent algorithm is called Back-Propagation ([27], [22]). First the error is calculated. The error function used here is calculated as follows:

$$E(t, y) = \frac{1}{2}\sum_{j=1}^{n}(t_j - y_j)^2. \tag{2.3}$$

Here t_j is the target-value (the value that should be the output), y_j is the value that the network has outputted, j is the j-th output neuron and n is the number of output neurons. After this, the partial derivative with respect to output neuron j is calculated for each output neuron. The output layer here is assumed to be using the identity function.

$$\delta_j = t_j - y_j. \tag{2.4}$$

Next, the derivatives of the hidden layer are calculated with respect to one hidden neuron, for every neuron, using the derivative of the logistic sigmoid:

$$\delta_i = y_i(1 - y_i)\sum_{j=1}^{n} w_{ji}\delta_j. \tag{2.5}$$

Here y_i is the output of hidden neuron i. After this the weights are updated by using δ_j for weights between the hidden layer and output layer and δ_i for weights between the input layer and hidden layer:

$$w_{ji} = w_{ji} + \alpha\delta_j x_i. \tag{2.6}$$

Here α is the learning speed, this is needed because otherwise it will only learn to map the input to the output of that certain learning moment. In other words it will constantly overfit. To solve this, α is set to a small value to make a small step into the steepest direction of the search space given the input and target values. Also equation 2.6 is used for all connections, only with a different δ. FNNs are often used in the form of supervised learning. Supervised learning means that when a FNN predicts something, the target value that is going to be learned, is given. This is in contrast with Reinforcement Learning. Here the target value is not given, but based on rewards the FNN gets for predicting the correct or incorrect action.

2.2 Time-series and Recurrent Neural Networks

Time-series data is data that is divided into discrete timesteps. This is because a computer splits up time into discrete timesteps, the data thus has to adapt to the system that is used. Furthermore the input is transferred into a number format which is a representation of the real world data. For example sound-waves can not be directly used as input, they are first transformed into a format that computers can deal with. In this case it is the frequency of the sound-wave. The time-series data thus consists of a matrix where each column can be seen as an input pattern at one timestep.

Time-series data can form a problem for machine learning algorithms. Normally an input pattern does not have important relations with other input patterns. In other words, one input pattern is not dependent on other patterns. In time-series data, temporal dependency of patterns can be important. For example, in recognizing sounds, the algorithm should store previous parts of sounds, otherwise it will only classify a sound based on the input at that timestep. To solve this, the algorithm needs a short-term memory. This forms a problem because, it is not clear which part of the information should be stored and how long something should be remembered. In a simple problem this might not be too hard to solve, but in most problems, where there is a lot of noise and other information that needs to be filtered out, this can reduce the transparency. Learning also forms a problem, because it is not necessarily known what needs to be learned and when this should happen. One solution to this problem is called a Liquid State Machine, here instead of learning the patterns with the short-term memory reservoir, the Liquid State Machine uses a readout network to learn to associate the states of the reservoir with the desired outputs.

Feed-forward Neural Networks have problems with time-series, as they are static and do not have any short-term memory. One solution is to buffer the input for a number of timesteps and giving this set as one input to the neural network. These networks are called Time-Delay Neural Networks (TDNN; [11]). This shows exactly the problem of for example FNNs. First the number of input neurons can be high (because of the long input patterns) which increases the chances of overfitting and training will take much longer than normal. Another problem is that for example with words, one word might be long and the other short, this means that some input patterns do not use the complete network, which makes learning more difficult. Also, the time-dependencies are unknown and can differ in length. A better approach are Recurrent Neural Networks (RNN). These come in a number of forms. One is based on an FNN, where activation is calculated by a sigmoid function. The difference between FNNs and RNNs is that RNNs have recurrent circuits. These circuits send output back into the network in the next timestep. This means that activation from the last timestep can be taken into account for calculating the next timestep, if there is a recurrent circuit in the network. An advantage in comparison with TDNNs, is that the short-term memory that TDNNs have, is only within a certain time-frame. It can be that some of the input of earlier timesteps could influence the input of later timesteps, but TDNNs do not take this into account, they just cut the time-series in pieces. RNNs do not have this problem as an input can be theoretically remembered forever, thus input from the beginning of the time-series, can be used for input at the end of the time-series. However, a drawback to RNNs is learning. There are learning rules based on the Gradient Descent algorithm, two well known Gradient Descent algorithms are: Real-Time Recurrent Learning (RTRL; e.g. [20]) and Back-Propagation Through Time (BPTT; e.g. [28]). In BPTT and RTRL the error is propagated back into the network over a number of timesteps. However, using Back-Propagation has a drawback. When propagating back further in the network, the contribution of lower lying connections to the error is getting smaller. It makes it difficult to get the importance of states that lie further back in time. This makes it hard to learn and also learning will be a lot longer in comparison with Feed-Forward Neural Networks. Mostly states that lie more than ten timesteps back in the past, can not reliably be used. This is also known as the the problem of the *vanishing gradient* ([7]; [1]; [17]; [28]; [12]).

Another implementation of RNNs are Spiking Neural Networks (SNN; [26]). SNN neurons, in comparison with neurons using a continuous function, are more inspired by biological neurons than most RNN neurons, as they share the property to spike. Although biological neurons are far more complex, they have some similarities. This of course partially depends on the implementation of the spiking neuron, as there are a lot of different types of neurons, in both biology as in computer science.

Most neurons in our brain only pass through a signal when the activation has passed a certain threshold, which causes an action potential. Because of this action potential, a signal is sent to the connecting neurons. To translate this to computational methods, it means that a neuron either sends a spike in a timestep or does not send anything at all. This differs from continuous networks, where there is almost always activation that is passed through. Another similarity between biological neurons and spiking neurons is that they both have an internal activation within the neuron that stays over time. This activation comes from input signals that are received some timesteps ago, and thus gives a short-term memory for each neuron. The activation also 'leaks' away, which influences the time-frame from which input should still be used (with a higher decay, activation leaks out faster and thus the neuron has a shorter memory). If the activation is low, one spike will probably not get the neuron to fire. But if at the same moment there are a number of different spikes that come into the neuron, it will increase the chances of the neuron to fire.

The problem with SNNs is that it is even more difficult for a SNN to find a good supervised learning method then it is for continuous RNNs. Although algorithms exist like SpikeProp [3], it is a computational heavy task to learn with Back-Propagation methods. Most learning methods in the brain seem to be self-organizing. In self-organization there is no feedback when a wrong value is outputted. The only data the self-organizing algorithm has is the input. One well known self-organization method is called Hebbian Learning [6]. This learning rule comes in many forms but they all are based on the same idea. That is that when two neurons are connected, and both neurons fire at the same time (or in the same time-frame), the connection between the two neurons gets stronger. If the neurons do not fire at the same time, the connection between the two neurons will be weakened.

2.3 Long Short-Term Memory

Long Short-Term memory networks (LSTM; [8]) are used in Evolino, which is described in section 3.4. Evolino optimizes these networks. A Long Short-Term Memory network is a Recurrent Neural Network that uses special neurons to calculate the output. These neurons are called memory cells and have gates to decide whether activation is flowing through the core and out or not, or some of it. There are three different types of gates: input gates, forget gates and output gates. The core has a recurrent circuit (in other words, the activation of one time-step back is part of the input for the core for the next time-step). This means that it can possibly remember an input infinitely long. The forget gate can 'leak' the activation out of the cell thus removing information that is not needed anymore. The input gate is there to protect the cell from input that is not needed. In other words, if the gate is closed, no input will come into the cell. The output gate decides when information should be sent over to other memory cells. Figure 2.3 shows the setup for the memory cell. The memory cells are connected in four ways, all output connections of a memory cell are connected through the output gate. The input connections (both from input neurons and other memory cells in the network) are connected through the core and the three different gates. The internal unit holds the state of the cell. The cell state is calculated as follows:

$$s_i(t) = net_i(t) g_i^{in}(t) + g_i^{forget}(t) s_i(t-1). \tag{2.7}$$

Here $s_i(t)$ is the state of cell i at time-step t, g_i^{in} is the input gate of cell i, g_i^{forget} is the forget gate of cell i, $s_i(t-1)$ is the state of cell i one time-step back and $net_i(t)$ is calculated as follows:

$$net_i(t) = \sum_{j=1}^{n} w_{ij}^{cell} c_j(t-1) + \sum_{k=1}^{o} w_{ik}^{cell} u_k(t). \tag{2.8}$$

Here c_j is the output of memory cell j, n the number of hidden cells, u_k the output of input cell k, o is the number of input cells and w is the weight between two cells (where cell j and k are input for cell i). The output of cell c_j is calculated as follows:

$$c_j(t) = tanh(g_j^{out}(t) s_j(t)). \tag{2.9}$$

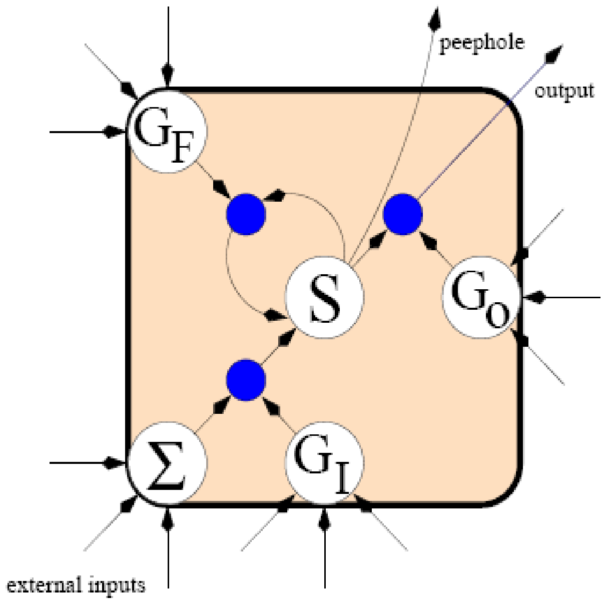


Figure 2.3: Figure of a memory cell (from [23]), S is the internal state of the cell, G_F is the forget gate, G_O is the output gate and G_I is the input gate. The Σ is the input from input cells and hidden cells. The blue nodes represent the multiplication functions

Here g_j^{out} is the output gate of cell j. The value of the gates at time t is calculated as follows:

$$g_i^{type} = \sigma(\sum_{j=1}^{n} w_{ij}^{type} c_j(t-1) + \sum_{k=1}^{o} w_{ik}^{type} u_k(t)), \tag{2.10}$$

where σ is the logistic sigmoid function and w^{type} is the weight for either the input, hidden or output gate. The strength of this model is that it has both long-term as well as short-term capabilities. Long-term storage is done by learning the weights between the cells. The short-term memory is created by the architecture of the memory cell and the wiring between the different cells. First there is the recurrent circuit for the core, but also the different gates add dynamics to the memory. Normally for learning a variant of Real-Time Recurrent Learning is used, but this will not be used in Evolino (the implementation can be found in [8]) to optimize the weights.

2.4 Liquid State Machines

2.4.1 Theory

Liquid State Machines (LSM; [14]) are a new concept in machine learning. They solve time-series problems in a completely different way in comparison with most RNN systems and go around the problems that RNNs often have with learning them. LSMs use a dynamic reservoir or liquid (L^M) to handle time-series data, as can be seen in figure 2.4. After a certain time-period, the

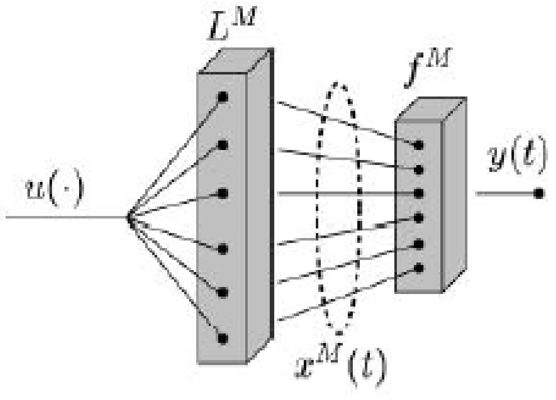


Figure 2.4: Figure of a liquid state machine (from [14]), where $u(\cdot)$ is the input for the liquid, L^M the liquid filter, $x^M(t)$ the liquid state at time t, f^M the readout network and $y(t)$ the output at time t

state of the liquid $x^M(t)$ is read out to use as input for a readout network f^M (for example a FNN). This readout network learns to map the states of the liquid to the target outputs. This means there is no need to train the weights of the RNN, which decreases the computation time and more importantly, the complexity of learning time-series data. A liquid can be represented in different forms, it can be a real liquid, where the waves can be seen as a short-term memory. For example, if someone would throw a rock into a lake, the waves that this rock creates, are the memory of the liquid. In other words, the waves tell that something has happened a short time ago. But a Spiking Neural Network (SNN) can store much more information than a real liquid. This is because in principle, a real liquid can have information in the three spatial dimensions. But a SNN, can have far more neurons than three, thus be able to store more information. SNNs consist of neurons that are more biologically plausible in comparison with other RNNs. These neurons have an internal activation and only communicate with each other by either not sending anything (the output of a neuron is zero) or sending a spike (the output of a neuron is one). The neuron has an internal activation which leaks out over a time period, this is a short-term memory and the amount of leakage is the time it takes to forget an input. An important property of SNNs is that they also have information in the spike code. This spike code is an array of spikes in a certain time-frame. Spike codes can be seen as a sort of Morse-code, where there can be pauses (when no spikes are sent) or a 'bleep' (when a spike is sent). The use of this, is that the time when a spike arrives is also a form of information. Another feature of the spike neuron is that it keeps its internal activation, although it leaks away over time, this means that due to recurrent circuits, the neuron itself also has a short-term memory system. LSMs are a good tool for classification problems. In classification problems, the LSM should separate different inputs from each other and classify these. An example of this is classifying the composer of musical pieces (this is also used in the experiment, see section 4.2 for further details). The musical piece is input for the liquid and the readout network should then classify which composer has composed the musical piece. Classification used by the readout network is also described in section 2.1. For classification, the readout network needs to separate the different states from the liquid. Given that there are enough units in the liquid, it can create different patterns for each time-series pattern (see [14] for further explanation).

Liquid State Machines have a Separation Property (SP) and Approximation Property (AP) [14]. SP addresses the ability to separate two different input sequence from each other. This is important, because the readout network needs to be able to separate two input patterns to have a good performance. If two patterns look too much alike if they should not, the readout network can not differentiate between the two patterns and thus is not able to tell which pattern belongs to which class. AP addresses the ability of the readout network to distinguish two different patterns and transform the states of the liquid into the given target output.

2.4.2 Reservoir and Readout

For implementing the liquid an SNN is used. Here the neurons are set in a grid, neurons are connected by a chance also used in [14]. The only exception is that initialization for input neurons is different because they are initialized using a static chance. The input layer is a two dimensional layer which represents the spatial properties of the data. The dimensions of the input layer when the input is for example a chess board, are the same as the dimensions of that chess board. The spatial properties in the liquid itself are kept because of this. In other words, the liquid is built up by layers where the width and height are the same as the input layer. The depth defines the number of layers there are in the liquid. The chances for connecting two internal neurons with each other is calculated by first calculating the Euclidian distance ($D(i, j)$):

$$D(i,j) = \sqrt{(xcor_i - xcor_j)^2 + (ycor_i - ycor_j)^2 + (zcor_i - zcor_j)^2}. \quad (2.11)$$

In this equation i and j are two different neurons, with coordinates $\{xcor, ycor, zcor\}$. After this, the chance to connect two neurons is calculated, using the following equation:

$$p_{connect}(i,j) = e^{\frac{-D(i,j)}{\lambda}}. \quad (2.12)$$

In equation 2.12 the λ parameter is used to control the probability distribution. The chance to not connect it is then $1 - p_{connect}$. To calculate the chance of a neuron being inhibitory, the chance $p_{inhibitory}$, is multiplied by $p_{connect}$. The chance of being excitatory is then: $1 - p_{inhibitory}$. This implementation is also used by Maass et al [14]. In Maass [14], equation 2.12 is multiplied by c, here c is set to one. In [14] Maass et al explains that the distribution of connections is an important aspect of the liquid. Having too much connections between neurons that lie far apart decreases the performance. Having a lot of local connections and a few long connections seems to be the best solution. With this randomly initialized network, there are no direct recurrent circuits, which means that a neuron can not connect with itself. For simplicity, weights are set to one single value.

A leaky integrate and fire model [26] is used for the spike neuron. The neuron has a certain threshold (η). To get a spike, the activation ($a_j(t)$) of a neuron should become higher than this threshold. After a neuron fired, there is a refraction period (r_{period}). During this refraction period, the activation is in its resting state (r_{rest}), which is a constant. In this refraction period, the neuron can not fire and when the refraction period is over, it starts its activation at the resting state. The activation inside the neuron, is increased or decreased by incoming activation from other neurons, but the activation inside the neuron also 'leaks' away, this is done by multiplying the decay (d) with the activation of the neuron one timestep ago. This decay has influence on the capabilities for a neuron to have a short-term memory. Basically setting the decay to zero, the neuron has no memory and it also makes it harder to create spikes. Because with a higher decay, the activation is often already closer to the threshold. Setting the decay to one, means that every input (after the refraction period, or the startup of the liquid) is kept in the activation of the neuron. The internal activation of the neuron is calculated every timestep as follows:

$$a_j(t) = \sum_{i=1}^{n}(x_i(t) \cdot w_{ji}) + (d \cdot a_j(t-1)), \tag{2.13}$$

where $a_j(t)$ is the activation of neuron j at timestep t, x_i is the output of neuron i (Here the neurons are both neurons inside the liquid and input neurons) and w_{ji} is the weight between neuron i and neuron j, where j is the receiving neuron. $a_j(t-1)$ is the activation of neuron j one timestep back.

After the activation is calculated, depending on the threshold, the neuron will either spike or not, which gives an output of respectively one or zero. The spikes are summed up per neuron and at a certain time step over a period of time (this is called a readout moment ($x^M(t)$) and the readout period is called x^M_{period}), these summed spikes are, together with the activation of the neurons at that moment, passed through to the readout network. The number of readout moments can vary. First because samples can be of different lengths and thus can provide either more or less readout moments. Another reason is that the number of readout moments is dependent on the number of timesteps that spikes are summed, i.e. a shorter readout means there are probably more readout moments per sample. The samples are the time-series data, that is the input for the liquid. For example a musical piece is one sample. The sample is broken up in discrete timesteps, and the sounds can for example be broken up into frequencies. The frequencies at one timestep are then an input pattern for the liquid. This input is passed through by the input neurons. In the liquid, each input neuron passes either a zero or one through to the spiking neurons in the liquid.

To get readout moments the liquid must be warmed up (*warmup*). During this warmup period no spikes are summed. This is done because there is no activation at the beginning, which can influence the performance as it is hard to separate two different samples without much activation. After the warmup period there are, dependent on the settings, a number of readout periods. Reading out the states of the liquid is repeated until the end of the sample. After this, the readout moments are stored and labeled with a class.

If the liquid has encountered every sample, the readout network will be trained. An FFN (see section 2.1) is used to classify the samples. This FNN consists of an input layer, one hidden layer and an output layer. The hidden layer uses a logistic sigmoid function, while the output layer is linear. As the experiments are done on classification tasks, the number of output neurons is equal to the number of different classes. The target is one for the neuron with the index which equals the index of the class, otherwise the target value is zero. Every time the FNN is initialized, the weights are the same as with all the other initializations. This is done to keep the experiments transparent. It is possible that with random initialization, the network may find a good starting point. This increases the performance and the next time it finds a bad starting point which decreases the performance. To train the readout network, Back-Propagation is used and each sample is run through the network and after that, it is immediately learned, which is called online gradient descent [2]. This is done in epochs, which means that the complete dataset is learned in one epoch and is repeated a number of times. To increase the learning speed of the readout network, the input patterns are scaled between zero and one. This means that the relationship between all the readouts from the liquids stays the same, but the inputs now lie between zero and one. This is beneficial to the FNN as it increases the learning speed (smaller input means smaller changes in the weights when learning).

Chapter 3

Training Paradigms

In the next sections, some algorithms that have been explained in the previous chapter, are used for optimizing RNNs. First the performance functions that will be used in the experiments are described in section 3.1. In subsection 3.1.2 a new method to measure the performance of a liquid is described (namely the separation). After this, a number of new methods are described that will be used to optimize liquids in LSMs. The liquid optimization is partly done on the wiring. The wiring are the connections between different neurons, here there are three different types, namely an excitatory connection, an inhibitory connection and no connection. In liquids where the wiring is initialized based on distances of neurons, the wiring is important, because the weight is set to one default value for all connections. Thus finding a better wiring could improve the performance. Reinforcement Learning (RL) and Evolutionary Algorithms (EA) are described in respectively sections 3.2 and 3.3. These two algorithms could be compared, as they have an overlap in problems they can solve. The EA will also be used to optimize the weights in the liquid. This is because EA's are able to deal with real values. In the last section, a system that already exists will be further explained, namely Evolino [23]. Evolino uses Long-Short Term Memory networks as the liquid and Evolutionary SubPopulations (ESP) to optimize these.

3.1 Performance Measurements

3.1.1 Readout Performance

The most straightforward method to analyze the performance of a liquid, is to use the performance measure of the readout network. Here this measure is both used as a tool to analyze data learned by the LSM, but it is also used as a performance measure in optimizing the liquid. The performance measure implemented here, represents the percentage of correct classified samples. First the index of the output neuron with the highest value is picked as the classifier. If the index of the output neuron equals the target class, the outcome is one, otherwise it will be zero. This outcome will be called o_{ij}, where i is the i-th sample of class j. To calculate the performance P the following equation is used:

$$P = \frac{\sum_{j=1}^{c} \frac{\sum_{i=1}^{s} o_{ij}}{s}}{c} \tag{3.1}$$

Here c is the number of classes and s the number of samples in that class. To calculate the performance, the performance is first calculated for each class and then these are combined in the final performance measure. This is because the readout network can be biased by the fact that there are more samples from one class, which will make it look like the readout network performs well, but it is just biased to choose the class with the most samples.

This measure seems to be a good measure for liquid optimization, as it is stable and the outcome is transparent. However, the readout network needs to be trained to get this performance. That means more computation time.

3.1.2 Separation

The disadvantage of using the performance of the readout network for optimization is that it is not really a fast method, as the readout network needs to be trained and that takes up computation time. Also it is an indirect method to calculate the performance of the liquid (It is indirect because the performance of the readout network is measured and not the performance of the liquid itself). It would be better to have a measure which calculates the performance of the liquid, in a direct method. In other words, there is no neural network or other kind of statistical learning method necessary to find the performance of the liquid. The separation of a liquid should be able to fill in this task. Maass et al [14] used the separation to get a good view on how well a liquid can divide two different input patterns. To calculate this separation S, the distance is calculated between two different samples. For this the Euclidian distance is used, which calculates the distance between different readout moments:

$$S(s_k, s_l) = \sqrt{\sum_{i=1}^{n} (N_{ki} - N_{li})^2}. \tag{3.2}$$

In equation 3.2 s is a readout sample from the liquid and N_{ki} and N_{li} are the readout moments from neuron i of samples k and l and n is the number of neurons in the liquid. Here are a two different methods to use the distance to calculate the separation:

- $$S(s_{ij}, \ldots, s_{nm}) = S_{different}(s_{ij}, \ldots, s_{nm}) - S_{same}(s_{ij}, \ldots, s_{nm}) \tag{3.3}$$
- $$S(s_{ij}, s_{nm}) = \frac{\sum_{h=1}^{v} S_{different}^{h}}{\sum_{h=1}^{v} S_{same}^{h}} \tag{3.4}$$

Here S is the separation, s_{ij} is the j-th sample in the i-th class, n is the number of different classes, m is the number of chosen samples for each class, which normally is two, h is the h-th time that the separation is calculated (for total separation), v is the number of separations that are calculated and S_{same} and $S_{different}$ are calculated as follows:

- $$S_{same}(s_{ij}, s_{nm}) = \sum_{i=1}^{n} S(s_{io}, s_{ip}) \tag{3.5}$$
- $$S_{different}(s_{ij}, s_{nm}) = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=i+1}^{n} \sum_{l=1}^{m} S(s_{ij}, s_{kl}) \tag{3.6}$$

In S_{same}, o and p are different chosen samples from the dataset. For one separation calculation, there are two samples chosen from each class. These samples are both used in S_{same} and $S_{different}$. The idea behind equation 3.3 is that the distance between two samples from different classes should be as large as possible, thus giving it a positive influence on the separation value, and the distance between samples of the same class should be as small as possible, so if the distance is larger, the separation should be lower. Equation 3.3 however is not a good representation of the dataset. This is because it uses only a few samples and not the complete set. Maass et al [14] solve this by summing a number of separations. In equation 3.4 this is also applied. Also, here the relation is given between the distance of samples from different classes and samples from the same class. For example, if there are two different classes and there is no relation at all, the outcome is two. If the outcome is higher than two, there is a positive relation, which means that the distance of samples from the same class lie closer than the distance from samples of different classes. If the outcome is under two, then there is a negative relation and thus the samples from different classes lie closer to each other than samples from the same class. A nice feature of the separation is that it can also calculate the separation of one neuron. This is done by only calculating the distance between two neurons and not a whole network. This may give some insight in how different neurons perform. Also it could be used to filter out neurons that have a low separation. Some neurons may have a low separation value which could indicate that these neurons give the readout network a hard time at separating different samples from each other.

3.2 Reinforcement Learning

3.2.1 Introduction

Reinforcement Learning (RL; [25]) is an important learning method in machine learning. It is less demanding than supervised learning, where exact data is needed in advance. This may become a problem for certain implementations like in robotics, where data is complex and hard to get in advance. RL solves this problem by using a reward system to provide as a method for learning the optimal solution. This reward is given at certain moments and can be negative to penalize or positive to reward. The RL algorithm has an agent, which can for example be a robot that drives through an office. An agent can make actions, these can be all sorts of actions, for example in robotics an action can be to go forward. Each action an agent takes, gets some kind of reward afterwards. For choosing a next action, the agent chooses the most likely action by choosing the one which has the highest expected reward. In other words the action, that gets on average the highest reward, is chosen.

The agent learns in the same environment where it will be used in practice. This environment depends on what problem is going to be solved. It can for example be a maze, where optimal routes are learned to find the fastest route out of the maze. But it can also be a real world environment for example an office. Learning while also behaving in the environment is a big difference with supervised learning and also other learning algorithms. Here learning is more an act-react method instead of learning everything in advance. This makes the system more dynamic, because if the environment changes, it can use knowledge from the old environment to solve problems in the new one. But it also gradually learns how to improve its behavior in this new environment. While other methods have to learn the new environment before the agent can again be put back into use again.

Basically there are two types of reinforcement learning algorithms, one type is episodic, for example Monte Carlo sampling [25], where at the end of an episode, the sum of rewards is returned to update the state-value, for example in a game of chess, where it is hard to evaluate during the game if the moves that are made are good or bad. After the agent finishes a game it gets the reward and this can be seen as an episode. The other type of learning is where a reward is directly given when an action is done, these are mainly algorithms based on TD-learning [25]. This means that it can cope with problems that do not have a clear ending, for example in robotics where a robot is supposed to be running around doing jobs all day. Both Monte-Carlo sampling and TD-learning are methods which can be applied in problems where the environment is not completely known, these methods will thus try to predict the best action.

Both Monte-Carlo sampling and TD-learning can be implemented in two ways, using V-values or Q-values. The difference is that with V-values, actions are chosen based on a model of the environment, where the algorithm investigates the states and chooses the action that leads to the best state (in other words gives the highest average return). Q-values have state-action pairs, here there is no model needed, it chooses the action that probably returns the highest reward.

For choosing actions, a RL agent uses a policy. There are two types of actions, namely exploitation actions and exploration actions. Exploration means that the action that is chosen is not the most likely choice at that moment, but they are needed to search in the search-space so the agent will not get trapped in a sub-optimal local optimum. In other words, exploring means trying to find new local optima. Exploitation is choosing the best action and thus choosing the action which probably will return the highest reward. It is important to have a good exploration policy, as this will balance the time an agent invests in exploration and exploitation. A couple of well known exploration policies are the ϵ-greedy policy and the soft-max policy. With ϵ-greedy normally the action with the highest value is chosen, but there is a chance to explore and thus to choose another action. This is done by using the ϵ parameter, this parameter sets the chance to explore and exploit. It is a straightforward policy and not the best, as it could be possible that the policy needs to change over time, for example to explore less and exploit more. The soft-max [25] policy is completely different, it uses the Boltzmann equation [25] to calculate the chances for each action, which means that the chance to be chosen depends on the value of the V or Q-value. In other words, the bigger the difference between two V or Q-values, the higher the chance the action with the highest value will be chosen. This is a more fair method of choosing a new action, as the relation between values is better represented in the distribution of the probabilities for the V or Q-values. Learning can be done in two ways, either with on-policy learning or off-policy learning. The difference between these two is that on-policy learning updates the action it has chosen using the value of the next action, while off-policy learning updates chosen action using the value of the best next action.

3.2.2 Using Reinforcement Learning for Optimizing Liquids

The implementation used for optimizing the wiring of a liquid, is a multi-agent system. This means that the learning process is more complicated than normal, as agents should work together and certainly depend on each other. In other words, if one agent changes its state, it may influence the whole chain of connections that come after this connection, including itself.

The algorithm is an on-policy Monte-Carlo sampling. As described in section 3.2.1 Monte-Carlo sampling is episodic. This seems to be suited for liquid optimization, because one run through all the samples can be seen as one episode. Furthermore an on-policy method is used. The RL algorithm here, uses Q-values, the agent is in a certain state and given this state, the agent chooses its action, these actions are explained later in this section. As a policy, the soft-max algorithm is used. As already stated in the last section, the soft-max algorithm uses the Boltzmann equation to calculate the chances of an action to be chosen:

$$p_a = \frac{e^{Q_t((n_i,n_j),a)/\tau}}{\sum_{b=1}^{n} e^{Q_t((n_i,n_j),b)/\tau}} \tag{3.7}$$

Where τ is the temperature and $Q_t((n_in_j),a)$ and $Q_t((n_in_j),b)$ are different Q-values for the agent in states a and b between neuron i and neuron j, with neuron j receiving output from neuron i. An agent is a connection between two neurons. In this case the agent has three different states, namely the state to be on, to be inhibitory and the state to be off. As described in section 2.4.2, the wiring of an LSM with a random liquid, will be chosen using the distance. The chances to initialize the wiring, are also used as initial chances for the RL algorithm. This may give a boost to initial learning. To find the right Q-values given the different chances, the following equation is used:

$$Q_t((n_i,n_j),b) = \ln\left(\frac{e^{Q_t((n_i,n_j),a)/\tau}}{e^{-dist_a(x,y,z)/\lambda}} - e^{Q_t((n_i,n_j),a)/\tau}\right)\cdot\tau \tag{3.8}$$

Here $Q_t((n_i,n_j),a)$ is the default value which is predefined. The complete explanation on how equation 3.8 is found and how to apply it to find the correct Q-values, can be found in appendix A. The soft-max algorithm also has a disadvantage. It gives another parameter that needs to be tweaked. There is also a trade-off that needs to be made on how to set the τ parameter and also the default Q-value. If the Q-value is high, because of the exponent, it will increase computation time (calculating the exponent of two takes longer than calculating the exponent of one). If the temperature (τ) is increased, the exponent will be smaller, thus less computation time. But the probabilities for the different Q-values lie closer to each other. This increases the time spend on exploration. The temperature needs to be tweaked according to how much exploitation and exploration there should be. Too much exploration and the system will not converge to an optimum, too little exploration and the system converges to a sub-optmimum.

The agent can be in three different states, namely: $\{-1,0,1\}$. These states stand for the type of connections that can be made, namely a positive connection, no connection or a negative connection. To update the Q-value, the following formula is used:

$$Q_t((n_i,n_j),s) = \begin{cases} (1-\beta)Q_t((n_i,n_j),s) + \theta[R(t)] & \text{if } s = 0 \\ (1-\beta)Q_t((n_i,n_j),s) + \alpha[R(t)] & \text{otherwise.} \end{cases}$$

Where β is the decay, α and θ are the learning speed, s is the state where the agent is in, t is the time-step and R is the reward. This system is used because otherwise if the number of connections is below fifty percent, it will update the agents in state 0 too high. θ thus should always be set lower than α. The reward given is the performance of LSM. The reward will only be given if the performance of timestep t is higher than the performance of timestep $t - 1$.

3.3 Evolutionary Computation

3.3.1 Introduction

Genetic Algorithms (GA; [4]) are based on evolution. GAs do not use one solution and build further on that specific solution to improve the performance, but they use a population of solutions and properties of the individuals in the population to find the optimal solution. As with evolution, selection on individuals takes place. The difference is that with GAs selection is done under more strict rules. The population is always of the same size, and in most cases the number of selected chromosomes for the next generation is also the same. Next to that the method for selecting individuals is different. Also GAs are directed to solving a goal, using a predefined fitness function, while in biology this goal seems to be missing.

There are a number of different methods to select individuals from the population. Two well known selection methods are truncation selection [4] (also known as ranked-based selection) and tournament selection [4]. The first is more straightforward. It selects a percentage of individuals with the best fitness. Truncation selection is not only used in computational problems, but also in other problems, for example farmers use it to get as much milk from cows as possible. They look at the milk production of a cow and take the best cows to make offspring. Tournament selection is a bit different. It uses tournaments to select individuals for the next generation. This is done by randomly choosing a fixed number of individuals from the population and checks which individual has the highest fitness. This individual will be selected for reproduction. A major difference between the two selection methods is that tournament selection has a more diverse population. Truncation selection only selects the best individuals at that moment, but it is possible that parts of solutions that are not selected can still be used to improve solutions in the future. In tournament selection this is partially overcome because it is possible that a tournament is held between a number of really bad solutions which means one of these bad solutions is selected and thus creates more variety in the solution space. This has some relations with reinforcement learning policies, where it is important to find a good method to both explore and exploit. In GAs this problem also needs to be solved, so that an optimal solution is found and not a sub-optimal. Another method to increase the exploring capabilities of the GA is to increase the population size, this also has a disadvantage, it increases the computation time per generation. This gives a bit of a trade-off to how large the population should be, the computation time should be as short as possible, but it should be big enough that it contains a wide variety of solutions.

For producing offspring, crossover and mutation are used. Crossover is a method to combine the gene-strings of two or more parents into one to get a child. Gene-strings are basically strings which contain the information of the system that is optimized. Two well known crossover methods are one-point crossover [4] and uniform crossover [4]. One-point crossover (see figure 3.1 for illustration) slices the gene-string of the parents at the same point into two parts, and uses the first part from the first parent and the second part of the second parent to make a new gene-string (This can be extended to use the second part of the first parent and the first part of the second parent to create another child). One-point crossover can be extended to two-point crossover (or three-point etc.), where more points are picked to slice up the gene-string for recombination. An advantage of this type of crossover method is that it keeps a certain ordering in the gene-string and sometimes this can be useful to solve the problem. Uniform crossover does not use this ordering, it picks either a gene from the first or second parent for each gene in the gene-string (as is illustrated in figure 3.2). This means that the ordering is less important, but this can also be an advantage as the spatial property of the gene-string can also negatively influence the solutions. Mutation is a method to create diversity. Mutation is done by randomly changing a value in the gene-string to another value (within the search-space). These two methods are different as mutation brings new diversity in the population, while crossover tries to use the best properties of the solutions that are already created.

One of the difficulties in GAs is to find a good fitness function. This is important, the fitness function will define in which direction of the search-space the GA will search. If the fitness function is flawed, it will not optimize to the wanted result. There are all sorts of problems for finding a good fitness

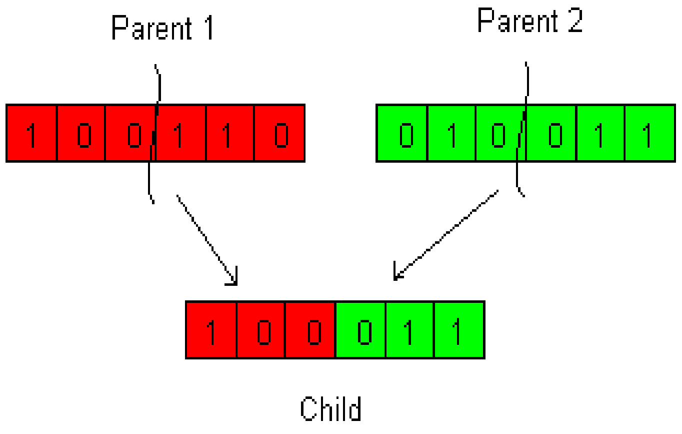


Figure 3.1: Illustration of 1-point crossover, the child inherits the first three genes from parent one and the rest from parent two

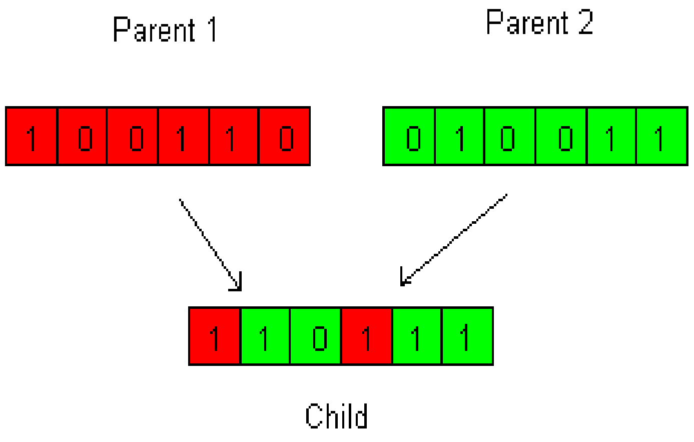


Figure 3.2: Illustration of uniform crossover, here the child inherits genes one and four from parent one and the rest from parent two

function. The function can be too complex, which results in difficulties to find an optimal parameter setting for the fitness function. An example of this is when an EA is used to design the skeleton of a car, it should be strong, but also light. Both the sturdiness and weight should be taken into account for the fitness function, but it is not clear how the combination should be made between the properties, which properties should be made more important. This increases the search-space and can result in a fitness-function that will not give the expected results. Next to that a fitness function can be biased. It can be that a fitness function is biased in a way that it ignores an important part of the search-space and thus the optimal solution will not be found.

3.3.2 Using Genetic Algorithms for Optimizing Liquids

Evolutionary algorithms play an important role in machine learning and a lot of the problems that are optimized using reinforcement learning are also optimized with evolutionary algorithms. It seems a good idea, to compare the performance of an evolutionary algorithm with that of a reinforcement learning algorithm. Other than that, optimizing the wiring of a liquid is much like the bit-string problems that are often optimized with genetic algorithms. Although Evolutionary Algorithms are already used to optimize Liquid State Machines and Echo State Networks, for example in [9], wiring optimization is not very common. The genestring is represented by two layers (see figure 3.3 for illustration), an input layer and the liquid layer. The input layer represents the connections from input neurons to liquid neurons and the liquid layer represents connections between two liquid neurons. Also unlike in random liquids, it is possible to have neurons that are connected to itself, because if this has a negative influence on the fitness, the GA will remove these recurrent circuits after a number of generations.

The algorithm that is implemented goes as follows:

1. The pool is initialized, wiring is randomly initialized as described in section 2.4.2. After this the layout of the wiring is put into a genestring as can be seen in figure 3.3.

2. Each chromosome is validated to calculate the fitness, by using the liquid in a LSM.

3. Parents are selected from the chromosome pool.

4. Offspring is produced by using crossover with two parents.

5. All chromosomes in the new pool have a chance to be mutated.

6. steps two, three and four and five are repeated until the end condition is met (here the end condition is the number of generations that is pre-defined)

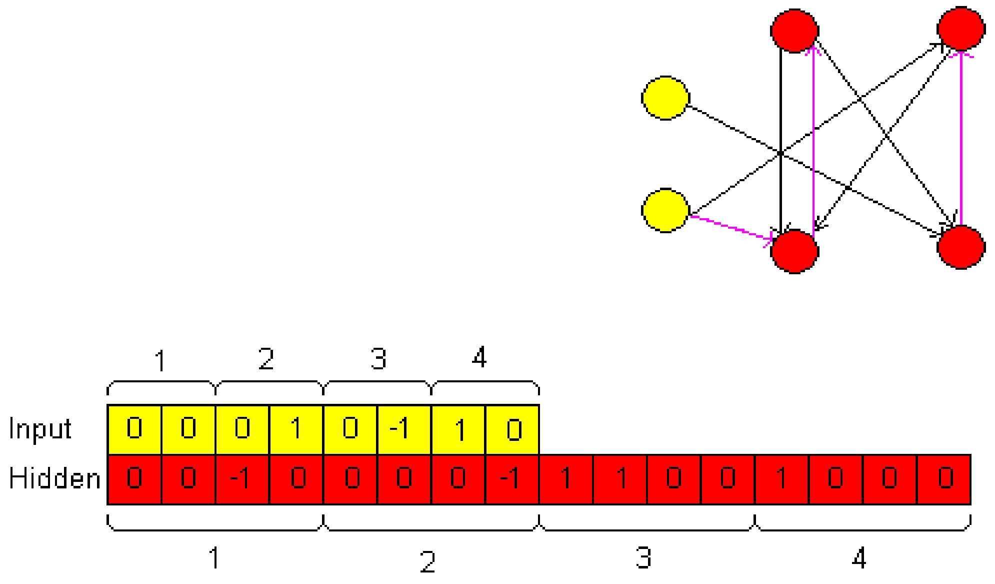


Figure 3.3: Illustration of how a liquid is converted to a genestring, the purple connections are inhibitory connections. Note that the representation of the liquid here is a two dimensional liquid, normally this would be a three dimensional liquid

For selecting the parent, tournament selection is used. In tournament selection a number of chromosomes are chosen to take part in a 'tournament', which means that the chromosome with the highest fitness wins and is selected for producing offspring. As already stated in 3.3.1, population size is important as is the number of individuals per tournament. More individuals in a tournament means that the chances are less likely that chromosomes with a lower fitness will be selected, thus there is a trade-off between choosing the best chromosomes and keeping the population diverse.

Selection takes place by shuffling the chromosome pool, and picking out the first number of chromosomes in the pool for a tournament. After the tournament, the best individual is put in the parent pool. This is repeated until the parent pool is filled. The chromosomes selected for a tournament are not put back into the pool. This way every chromosome gets a chance to be selected as a parent and be in the next generation pool (on the condition that it wins the tournament of course). A small change to this implementation can be made, by selecting a small number of the best chromosomes without using tournament selection. This way, the best chromosomes are kept in the gene-pool.

After that, the parent pool is shuffled and the first two parents are selected to make a child, this is repeated until the chromosome pool is full again. These parents will not be put back into the old parent pool, thus every chromosome has an equal chance to make offspring and most likely will (if the number of parents selected for the new pool is low enough every parent will have an offspring). Of course if the parent pool is empty, but there are still not enough children in the pool, the whole process is repeated again until the pool is full.

Children are made by using crossover. For the implementation that is used here, uniform crossover is used. This is because the ordering of the genestring is different of that of the liquid (the liquid is 3d and genestring is 2d), this should not affect the solutions that are generated. Also to make a one-point crossover (or multi-point crossover) that cuts a 3d slice in the genestring to combine with another parent, would be complex, thus the choice is to not use the spatial property at all. After this, all chromosomes, including the parents chromosomes have a chance to mutate, this is done by going through the genestring and with the chance p_{mutate}, a gene will be mutated. If the gene is mutated, there is a fifty percent chance to mutate in one value and 50 percent to mutate in the other. For example, if the gene has the value of one, it will either with a chance of a half, mutate in minus one, or zero. The parent pool is also mutated, because this reduces the chances of elitism. Elitism means that only genes from the best chromosomes are kept in the population, but it is possible that chromosomes with a lower fitness still have some good properties that could be used in next generations. Mutation already solves this by keeping the population diverse, but if it is not done on the parents, there is a chance that the diversity in the gene-pool is low and thus increases the chances of getting elitism. A disadvantage of this implementation however, is that the mutated parents get a different fitness value and this increases the computation time. An exception however are the parents that are selected without the use of tournament selection, as is explained above. These chromosomes will not be mutated as this would remove its purpose of at least keeping a small number of the best chromosomes in the gene-pool.

After mutation, the fitness is calculated for every chromosome in the pool (including mutated parents). Fitness is calculated by training the readout network on the training dataset. After this a control set is used to determine the performance of the LSM and this value is used as fitness.

As an extension another layer is added to the chromosome. This layer consists of the weights between the neurons. The weights and wiring are optimized together. It is faster to optimize weights parallel to optimizing the wiring, because the computation time is dependent on how many connections there are, and the wiring thus reduces computation time (given that there are connections missing between neurons). The representation is kept as in figure 3.3, but now there are four layers, namely two input layers and two hidden layers (one for weights and one for the connections).

The whole process of optimizing weights is the same as with the wiring, with one exception and that is mutation. To mutate a weight, a random value between -0.025 and 0.025 is picked and added to the original weight. Also a weight can not be below zero or above one. Because the wiring already has a negative value, a negative weight would make this positive and can interfere with the learning process. In other words, if a weight is negative and a connection is negative and the fitness is better because it is a positive number, when crossover is used this negative weight can be chosen in combination with a positive connection, which can have a negative influence on the fitness. A weight can not be higher than one, because a weight could increase to infinity which can be a negative influence on learning. Furthermore it could be more biologically plausible as a neuron only has a certain amount of energy and can not make the signal any stronger than that. The maximum of one is chosen, because this is also the threshold value and firing value of a neuron, so a signal can not become higher than its spike value. Using wiring in combination with weights has some disadvantages. Weights are selected while the connection at that gene could be in the off-state, so a weight that has no influence on the outcome, is still chosen in the next gene-pool. This can be negative but it can also be positive, as there is more diversity in the gene-pool.

3.4 Evolino

3.4.1 Introduction to Evolino

Evolino stands for EVolution of recurrent systems with Optimal LINear Output. Evolino has the same architecture as LSMs and ESNs, it uses a liquid and readout network. The liquid is a LSTM (as described in 2.3), which is then optimized using evolution as described in the next section (section 3.4.2). The readout network that is used is a linear regression model as the name Evolino (LINear Output) already implies. In section 3.4.3, the implementation is described that is used in the experiments.

3.4.2 Enforced SubPopulations

Enforced SubPopulations (ESP) is an evolutionary approach that is an extension to Symbiotic, Adaptive NeuroEvolution (SANE; [15]). It is a co-evolutionary approach to solving Neuro-Evolutionary problems. In most EA approaches, a chromosome represents the complete neural network, but in SANE individual neurons are optimized. These neurons in most implementations are basically a set of weights (real numbers) for that neuron. This is done by making a genepool of neuron chromosomes and combining a number of these to make a network. After this, the fitness of the network is calculated and each chromosome that took part in the network, gets that fitness. The difference between SANE and ESP is that in ESP there are sub-populations, which means that for each point in the neural network there is a sub-population that is evolved (see figure 3.4 for an illustration). SANE does not have different sub-populations, there is only one chromosome pool with neurons in it and each neuron can be chosen for each place in the network. This means that ESP has more specialized functions, which may increase the variety in the genepool. The algorithm runs as follows:

1. First the populations are initialized. Each sub-population is of the same size. Each chromosome consists of weights between that neuron and both the input and hidden layer. These weight are randomly initialized real values. In Evolino, this is extended for the memory cells, which consist of four different types of weights, namely the three gate weights and the normal cell weight. The total number of weights is then $4 \times (I + H)$. This is four times (for each gate and the cell weight) the weights between the input neurons and the memory cell and four times

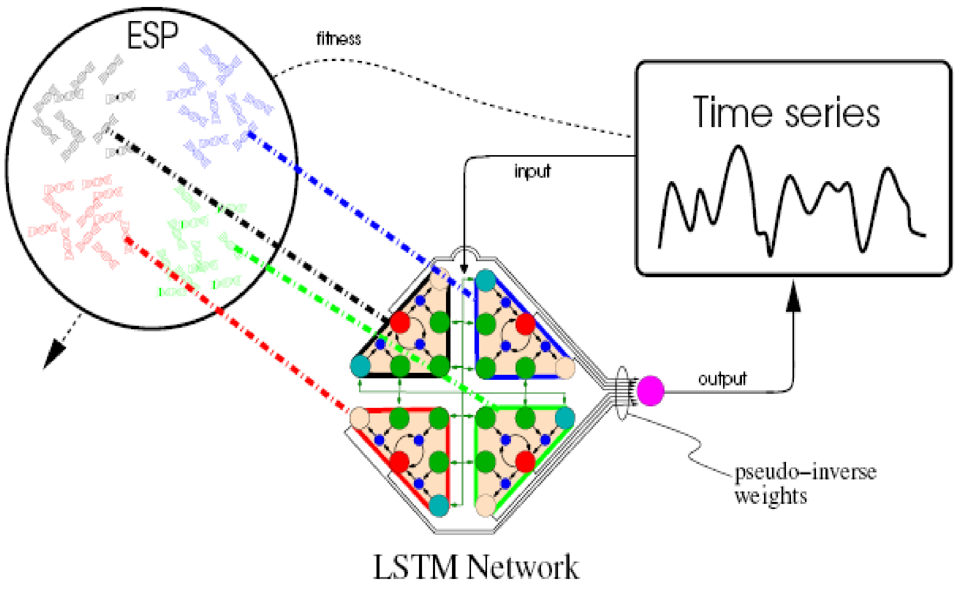


Figure 3.4: Illustration from [23] shows how ESP works. From each sub-population in the gene pool (on the left), a chromosome is taken, in the middle there is the LSTM network and the four chromosomes chosen from the sub-populations, are implemented in the network

the weights between hidden memory cells and the memory cell that is initialized here.

2. The neurons are evaluated. In each sub-population one neuron is chosen to combine into a network. Then the network is evaluated and gets a fitness score. This score is added to the *cumulative fitness*, which is the actual fitness that is used to rate the chromosome. This process is done for each neuron in each sub-population and repeated until each neuron has participated in a predefined number of evaluations.

3. Children are made to make a new generation. Dependent on the implementation, there can be crossover and mutation within a sub-population and selection methods can also differ. Most selection methods that are used in GAs can also be used in ESP.

4. Step two and three are repeated until the stop conditions are met. The

condition in Evolino is the number of generations it will train.

3.4.3 Evolino Implemented

As stated in the introduction, Evolino also uses a recurrent network to handle the time-series, but uses a linear readout network to map the states of the recurrent network to the output layer. The readout network is a linear regression model, which basically is the output layer of Evolino. It is trained with the Moore-Penrose pseudo-inverse method [18]. This method is chosen because it is known to be a fast and powerful method to learn the linear outputs. The weights in the LSTM are optimized by evolution, in this case, ESP is used to evolve the network. To evaluate the network (for the fitness), first all the samples are presented to the LSTM, after this, the output that is produced by the LSTM, is stored in a matrix to be learned by the linear regression model. To get the readout moments from the LSTM, first the LSTM is warmed up, which means that the output of the LSTM is not used. After that for a number of time-steps, the output is summed and used as input for the linear regression model. For all vectors in the matrix, there is a target vector, for training the regression model. After all samples are passed through the LSTM, the output weights are trained. After this, the training-set is presented again to the LSTM and now with the optimal regression model, the error that is calculated for the output units, is added to the fitness of the chromosomes in the network.

The memory cells that need to be optimized with the ESP which can be seen in figure 3.5, are represented by a string which contains all weight connections in the memory cell. These are the weights for both input neurons as hidden memory cells. The representation consists of the weights for the input, output and hidden gates and the cell weights that connect two units (either input units or memory cells) with each other. First the best twenty five percent of each subpopulation is selected. After this, to create offspring, both mutation and crossover are used. The crossover method that is used here is one-point crossover. The explanation about how this crossover works can be found in section 3.3.1. Mutation is done by adding some noise, created by using the Cauchy distribution [23]. The Cauchy distribution is calculated as follows:

$$f(x) = \frac{\varpi}{\pi(\varpi^2 + x^2)}. \tag{3.9}$$

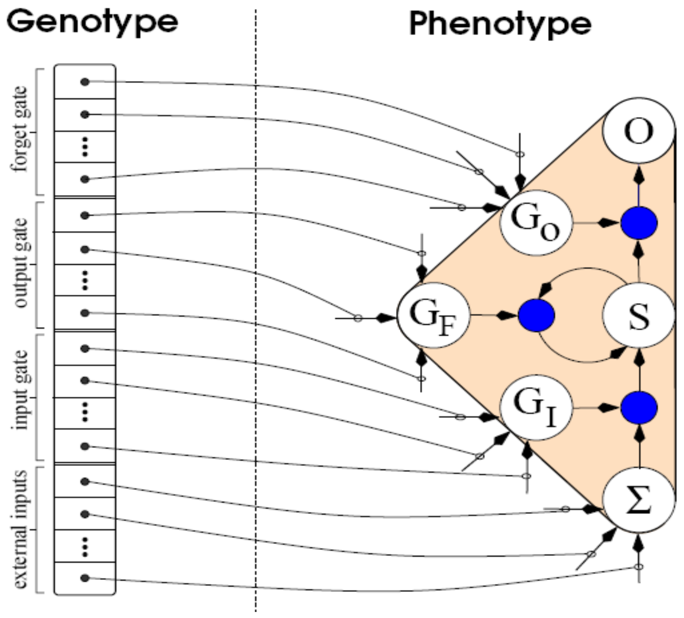


Figure 3.5: Illustration from [23] shows how a memory cell is represented in a chromosome.

Here ϖ is the interval of the distribution. After offspring is created, the fifty percent worst chromosomes in the sub-populations are replaced by offspring. If the performance of the ESP will not improve, burst mutation is used. This is used when for a number of generations, there is no performance improvement. Burst mutation keeps the best neurons in the population and the rest of the populations are filled with mutations of those best neurons. After this the algorithm resumes, but in a different point in the search space. This method can be repeated a number of times when there are no increases in performance. After ESP finishes training, the best memory cells of each sub-population are used to build a network, that is used for new data.

Chapter 4

Data

4.1 Movement Classification Task

It is difficult to find a good classification task to solve. First of all, most datasets do not contain a lot of data, because it often is hard to get a good representation in the data, if the data comes from a real world problem. This means that algorithms will not always perform the way they should do. If the set is too small, any algorithm will have a hard time finding the common patterns to divide the data into classes. This may give a bad view on how good an algorithm can solve the problem. The solution is to use a dummy dataset. The advantage is that there is more control over what is happening. The training set can be as large or small as required and it is easy to adjust a few parameters to make the problem to classify correctly, harder or a bit easier. Furthermore it is easier to have multiple datasets (like a test set and evaluation set). Because most real world datasets do not have that much samples, it will be even worse when it has to be divided into a training set and test set and maybe even an evaluation set for training a GA.

A problem however is finding a good problem-set to optimize. It should not be too hard or complex because then it does not give the transparency that is needed, but it also should not be too simple because then there are more simple methods to find an answer to the problem. A nice implementation is a movement classification task, a temporal dependency is needed to solve the problem and it is simple to add some noise to make solving the problem more difficult.

The set-up is as follows: there is a round-world grid (see figure 4.1). In

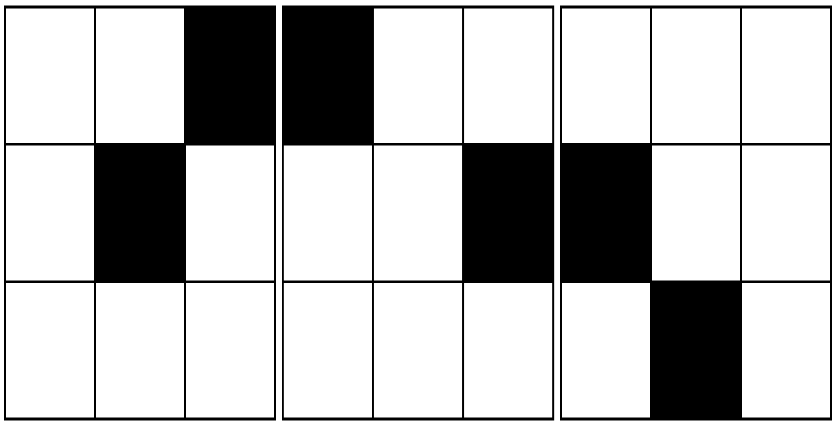


Figure 4.1: An example of a sample of three steps with the patch moving east, including grid noise.

other words, the left border is connected with the right border and the top border is connected with the bottom border, thus when a target moves of the grid, it will appear on the other side. This grid is built up of white squares and the size can be preset. Furthermore, there is a patch (or target) that can be of any size (as long as it is smaller than the grid). The patch is a black square and the patch either moves north, east, south or west. The task for time-series algorithm is to classify which direction the patch is moving. The input pattern is simple, every point in the grid is an input and if the patch is not on a point, it will give a zero as input, if the patch is at a point it will have a one as input. To use this as input, it is transposed to a one dimensional string as in figure 4.2. This is what is used as input for the RNN. Furthermore to make this problem more challenging, noise can be added. If the patch is not on a point, that point has a probability that it is a noisy input. A square that is noise will give an input of one in the input string. Another method to increase the complexity is changing the number of samples in a dataset, decreasing the number of samples will make it harder to solve the problem, but it gives more insight on how much the RNN will generalize to solve the problem.

1.	0	0	1	0	1	0	0	0	0
2.	1	0	0	0	0	1	0	0	0
3.	0	0	0	1	0	0	0	1	0

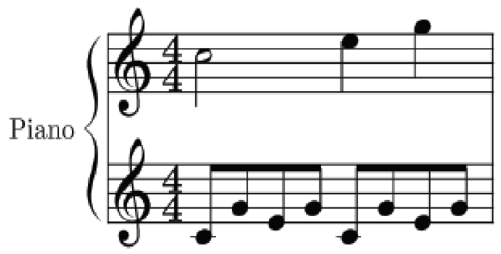
Figure 4.2: representation of figure 4.1 as input for the liquid

4.2 Music Classification Task

Music classification can be a nice tool to assist an user in recognizing musical pieces. For example, there are databases containing music that needs to be ordered. Because these databases are so big, it may be difficult to only let human users recognize the music. A machine can do this faster. In the experiments, the task is to classify the correct composer for a musical piece. The dataset is taken from the experiments done in [16]. The dataset contains pieces of Johann Sebastian Bach namely the Well Tempered Clavier and Ludwig van Beethoven with the Piano Sonatas. Although Bach is from an earlier time than Beethoven, there are similarities. Beethoven has played and studied the music of Bach and is also certainly inspired by Bach's music.

There are two methods of storing music in a digital form. The first is in waves, this is a digital recording of real (analog) instruments. The other method is to store information about the musical piece instead of storing the wave data generated by musical instruments. This contains information about notes, about volume, pitch, onset and length. The sound of the instrument is not stored. A well known digital format that uses this is called MIDI. An advantage for the learning algorithm is that it contains less noise in comparison with recordings, for example striking a string on a guitar is always different which results in a different soundwave and thus makes it harder to recognize as the same. As with the wave format, the learning algorithm first has to transform the sounds into something useful. Notes in MIDI files do not need that kind of computation as it contains the information needed. Thus in the experiments the musical pieces will be presented in MIDI form.

There is however some filtering, because MIDI files may contain subjective data. The only thing that needs to be recognized is which composer made the musical piece, but a musician can give his or her own feeling to it, with respect to the volume of notes, the timing, velocity. This needs to be filtered out. A solution is to remove the information about volume and about the exact onset and duration of the note. The length of a note is divided into timesteps of intervals 1/32 note. Smaller notes will not be used. Every piece will be transformed into the same key. This makes it impossible for the algorithms to classify only using the key of the piece. For pieces in a major key this is C-major and for pieces in a minor key, this will be A-minor. Normally MIDI files can contain 128 different pitches, but because everything is played on a keyboard, a number of notes are removed (as keyboards normally do not have 128 keys). After pre-processing, the musical piece is transformed into a time-series data set. The notes are transformed into discrete timesteps of the length of a 1/32 note. For example an 1/16 note is two 1/32 notes in two following timesteps. Each timestep consists of 72 numbers that will be used as input for the RNN. These notes correspond with 72 keys on the keyboard starting with f2. Every key can either be on (input will be one) or off (input will be zero). In figure 4.3, the representation can be seen.



$$\begin{array}{ccccccc|c|}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}$$

$t \longrightarrow$

Figure 4.3: On the top the representation of a MIDI file in staff and under it, the representation for the RNN. The numbers in the dashed box represent one input pattern for the RNN in one timestep. The figure is taken from [16].

Chapter 5

Experiments and Results

In this chapter, the setup and experiments that have been done will be discussed. In section 5.1, the parameter settings that are used for the different systems are explained. These parameters can be found in a number of tables. The experiments that are discussed will give some insight in the thesis questions. The first experiments that are done, are about the separation function. The separation is an important part to investigate. This is because the separation function can decrease computation time quite a bit in comparison with the use of a readout network. But the most important aspect of the separation function is that the results should correlate with the performance of a trained readout network. If this is not the case, it is useless to use the separation property as a fitness function, or reward function in the optimization algorithms. Because it is the readout network that gives the final performance. In other words, suppose with a sample, the separation is high, but the performance of the readout network is low, the separation will not tell how well the complete Liquid State Machine performs. A side experiment that can be done using the separation is using individual neuron separations. Some neurons may increase the performance of the liquid, but the readout states of those neurons can be of a negative influence as an input for the readout network. Removing some neurons from the input pattern for the readout network could increase the performance of the LSM. Although if the first experiment does not have positive results it will be unlikely that this experiment will perform any better.

The second part of the experiments is about training the liquid. These experiments will show how well the training algorithms will improve the liquid, in comparison with a randomly initialized liquid. This is done on the movement classification dataset. Furthermore the Genetic Algorithm is also used on the music classification data. Another experiment that is done here is using the datasets with Evolino. This is important, because it will give a good view on what the potential is of the LSM, given the type of liquid.

In the experiments the outcomes are shown graphically. There are a number of different graphs. There a number of bardiagrams. These make it easy to compare a number of parameter settings to each other. Furthermore some graphs show a spread of the data. This shows the relation between the separation function and performance function of the readout network. Some data is shown in a line graph which is used for example for Genetic Algorithms to show the learning curve. For comparing the systems with each other, the rest of the data is shown in boxplots. This shows the distances between the quartiles and also the average is plotted in the boxplots. Some graphs show the outcome of the training set, test set and evaluation set. The training set is the set where the readout network is trained on. The evaluation set is the set which is used as an performance measure in the training paradigms (except for Evolino, which uses the training set for this). The test set is used to measure the overall performance of the system. All performances are averages over a number of experiments and with the training paradigms, only the best solution for each experiment is used. The results are shown in percentage of correctly predicted readout samples. Note that with the music classification task there can be several different readout moments, thus having several predictions on one musical piece. A final remark to make is that there is a horizontal red line in the different graphs. This line shows the performance of a random generator.

5.1 Setup

5.1.1 Setup Datasets

The parameters for the movement task can be found in table 5.1. The size of the grid is nine. This size is chosen because the task is to find out which algorithm performs best, having a larger grid only makes training take longer. Also because of the small grid, the patch size is also small. In the training, test and evaluation set there are 150 samples. This is because a sufficient number of samples is needed to be able to compare the different algorithms. The length of a sample should be short, as it is easy to classify the direction of a target with a long sample (humans often need only a short amount of time to classify a direction).

Parameter description	Value
Number of samples	150
Height	3
Width	3
Patch size	1
Sample length	15

Table 5.1: Parameter setting for movement input

The parameter settings for the music classification task can be found in table 5.2. The total number of samples of the Bach-Beethoven dataset is 99 with 49 for Bach and 40 for Beethoven. Note that this is not the number of samples for the readout network, because each sample consists of multiple readout moments. The average number of timesteps in the Bach set is 2112 and with Beethoven this is 3822. The training set consists of 19 samples from Bach and 20 samples from Beethoven. In both the test set and evaluation set there are 17 samples from Bach and 15 from Beethoven. These numbers are chosen because the training set needs a bit more samples, this is due to the small sample size, the readout network that is trained needs enough data to train on to find all the specific patterns to differentiate between the two composers. The average length of a sample is 2976, the minimum length is 672 and maximum length is 9580. Furthermore as already stated in the last chapter, each timestep, that is input for the recurrent reservoir, consists of 72 points (12×6). In other words, six octaves. Although an octave consists of only eight notes (octave is derived from the Greek okto which means eight), in western musical theory, including the base note of the octave there are twelve notes between the first note of the first and next octave.

Composer	# samples	Average # timesteps
Bach	49	2112
Beethoven	40	3822

Table 5.2: Parameter setting for movement input

5.1.2 Setup Liquid State Machine

Each experiment with LSMs is repeated a number of times. For a random initialized liquid all the experiments are done ten times. For a trained liquid each experiment is repeated ten times on the movement classification task and five times on the music classification task. By repeating the experiments, the results will give a more reliable view of the performance in comparison when using the performance of only one experiment. This is because if for example a liquid is randomly initialized, the performance can be high, but this does not mean that every randomly initialized liquid will perform that high. It thus removes the bias or at least lowers the bias when repeating an experiment a number of times.

The parameter settings of the LSM for the movement classification task can be found in table 5.3. As can be seen in table 5.3, the liquid consists of twelve layers of nine neurons. From a small experiment it showed that twelve layers should be sufficient, the results can be found in figure 5.1. Furthermore because of the short sample length, the decay (d) is high, because every neuron should take as much of the earlier time-steps into account. The resting state (r_{rest}) is calculated as follows: $-(decay * r_{period})/10$. Here r_{period} is the refraction period, which is here set to one as the sample is short and it would lead to little activity if this was set high. The warmup period ($warmup$) is chosen to be set half of the length of the readout period (x^{M}_{period}), this is respectively five and ten, which sums up to the length of one sample and thus gives one readout moment per sample. λ, which is used for the chance to connect two neurons is set to 1.058, this parameter is found by using a Genetic Algorithm. The parameter for the chance of a connection to be inhibitory is also found by a GA. The threshold (η) for a neuron to fire is set to one, which is equal to the value of an output signal of a firing neuron. For random initialized liquids and for the Reinforcement Learning algorithm the weight is set to 0.4. The reason is that it is difficult to find a good heuristic that can set the initial weights in such a way that it improves the performance.

The parameter settings for the music classification task can be found in table 5.4. Most of the parameters that are used in the experiments for music classification are taken from [16]. One difference is the number of neurons in the liquid, this is decreased to 144, because it takes up less computation time.

The settings for the readout network for the two datasets are partially

Parameter description	Symbol	Value
Number of input neurons		9
Number of liquid neurons		108
Width		3
Height		3
Depth		12
Decay	d	0.95
Threshold	η	1
Refraction period	r_{period}	1
Resting state	r_{rest}	-0.095
Warmup period	$warmup$	5
Readout period	x^{M}_{period}	10
Connectivity parameter	λ	1.058
Weight		0.4
Chance of inhibitory	$p_{inhibitory}$	0.486
Chance to connect input	$p^{input}_{connect}$	1

Table 5.3: Parameter setting for liquid in movement task

the same, the parameters can be found in tables 5.5 and 5.6. The learning speed (α) is set to 0.025 for both sets, this showed to be a sufficient learning speed. The number of epochs for the movement task is set to 1500, because the readout network is converged at that moment. This is also the case for the music classification task, which is converged at 750 epochs. The weights between the nodes are randomly initialized between -0.25 and 0.25.

The parameter settings for the Genetic Algorithm can be found in table 5.7. The number of generations differs between the two datasets. For the music classification task this is set to fifteen, because it showed that after fifteen generations there is no improvement in performance. For the movement classification task, there are two settings, for a noise rate of 0.01 and 0.05 the number of generations is set to 50 and for a noise of 0.1 it is set to 30. A reason for this is that the results showed that the population was almost converged and the other reason is to decrease computation time. As a selection method, tournament selection is used, which has tournament sizes of five. This number seems to give a reasonable selection pressure, while still holding enough diversity in the population. Furthermore both the wiring and the weights are optimized during training.

Parameter description	Symbol	Value
Number of input neurons		72
Number of liquid neurons		144
Width		12
Height		6
Depth		2
Decay	d	0.8409
Threshold	η	1
Refraction period	r_{period}	4
Resting state	r_{rest}	-0.336
Warmup period	$warmup$	64
Readout period	x^M_{period}	128
Connectivity parameter	λ	2
Weight		0.4
Chance of inhibitory	$p_{inhibitory}$	0.35
Chance to connect input	$p^{input}_{connect}$	1

Table 5.4: Parameter setting for liquid in music task

5.1.3 Setup Evolino

Both the movement classification and music classification datasets are used with Evolino. The parameter settings for both datasets are a bit different and can be found in tables 5.8 and 5.9. The LSTM network uses twenty memory cells for the movement classification task. This is significantly lower than with the Liquid State Machine setting used above. But enough to get a good performance on the set. The subpopulation size for the ESP algorithm is set to thirty. The populations do not have to be as large as with Genetic Algorithms, because of the large diversity in the subpopulations. One remark to make is that using the dataset with noise of 0.01, no crossover is used. Because of the high computation time on the music classification set, it is hard to find optimal parameter settings. Because of this, the parameters are set to get a reasonable performance, but with a limited computation time.

Parameter description	Symbol	Value
Number of input neurons		108
Number of hidden neurons		8
Number of output neurons		4
Learning rate	α	0.025
Epochs		1500

Table 5.5: Parameter setting for readout network for movement task

Parameter description	Symbol	Value
Number of input neurons		288
Number of hidden neurons		8
Number of output neurons		2
Learning rate	α	0.025
Epochs		750

Table 5.6: Parameter setting for readout network for music task

5.2 Results

5.2.1 Separation

The separation function that is used in these experiments can be found in equation 3.4. The first experiment done is investigate how many times the separation should be summed to get a stable result (v). This experiment is done on the movement classification set with a noise rate of 0.01 and is repeated ten times. There are a number of different sums for the separation plotted in figure 5.2. The outcome that is plotted, is the outcome on the testset. To get an idea what should be a stable number, the variation in respect to the separation should be low. For example with $v = 10$, there is a high variation in the spread. The variation already decreases quite a lot with $v = 50$. But to be sure of a really stable function, the choice is to use $v = 1000$ in the following experiments.

If the separation function is used as an evaluation function for the optimization algorithms instead of the performance of the readout network, the outputs of the separation function should correlate with the performance of the readout network. This is because, if the separation is high, but the performance of the readout network is low, the liquid with the good separation

Parameter description	Value
Percentage parents selected	40%
Tournament size	5
Mutation factor	0.001

Table 5.7: Genetic Algorithm

Parameter description	Symbol	Value
Memory cells		20
Generations		75
Subpopulation		30
Cauchy distribution parameter	ϖ	0.005
Number of times fitness calculation		1

Table 5.8: Parameters Evolino movement dataset

will be chosen. The total Liquid State Machine will not increase its performance, but may even decrease its performance. Although figure 5.2 already showed some results, figure 5.3 gives a better overview on the performance of the separation. The experiments are done on ten different randomly initialized liquids, using the movement classification set with a noise of 0.01. In the results there seems to be little correlation between the separation and the performance of the readout network. Taking the test for example, two low separations have the highest scores in performance on that set. This graph should have shown an almost linear line, where higher performance also gives a higher separation. The only positive point to make is on the training set, because here a higher performance also gives a higher separation, although there is variation. Also, the training set is not the important set to compare with, as this is where the readout network is trained on and is thus a bit biased to it. A reason for this disappointing result could be that the readout network is non-linear function and is thus able to separate data in a different method than the separation function does. The conclusion of this result is that the evaluation function of the training paradigms, for example the fitness function of the Genetic Algorithm, will be the performance of the readout network and not the separation function.

The last experiment done using the separation function, is investigating at what the performances are when filtering the neurons readout for the input of the readout network. As already described in section 3.1.2, the idea is that

Parameter description	Symbol	Value
Memory cells		21
Generations		40
Subpopulation		30
Cauchy distribution parameter	ϖ	0.005
Number of times fitness calculation		1

Table 5.9: Parameters Evolino music dataset

neurons with a low separation can have a bad influence on the performance of the readout network. To solve this, only a percentage of neurons with the highest separations are used in the readout network. Experiments are done on the movement classification task with 0.01 noise, with LSMs using ten percent of the neurons for the readout network up to using a hundred percent, with steps of ten percent. For each percentage, the experiment is repeated ten times. The results are shown in figure 5.4. It is clear that if neurons are filtered in any percentage, it will not perform better than when using the complete network. One reason is probably the fact that the separation and performance of the readout network do not correlate that well. Another reason is that although neurons may have a low separation, all the neurons together carry more information than with some filtered out.

5.2.2 Training Paradigms

The first experiments to investigate, are the experiments using the movement classification task. This is a *toy problem* and thus gives more control over the set, which makes it easier to compare the different systems.

The results of Reinforcement Learning are not included, a number of different parameters are tried, but the results did not show any improvements to the initial performance. Thus, the experiments for optimization of the liquid in an LSM, are done using a GA.

On the movement classification task, the different algorithms are all trained using three different noise rates on the movement patterns, namely: $\{0.01, 0.05, 0.1\}$. The average performances of the best chromosome for the GA are plotted in three different line graphs and can be found in figures 5.5, 5.6 and 5.7. These plots show the curve of a chromosome trained using the evaluation set and also shows the performance on the training set and test set.

In figures 5.8, 5.9 and 5.10 all the average results are plotted for the test set, which are marked with a red +. This is done for a trained liquid using a GA, a randomly initialized liquid, Evolino and also for an FNN that does not use a recurrent network. Using only an FNN works in the same way as with a LSM, only instead of running the data through a recurrent network, it is directly fed to the FNN. There is a warmup period, where no input is used, after that input is summed until a readout moment. The summed values are than the input for the FNN. Using only an FNN, gives a good view on how well a recurrent network can change the representation of the data for the readout network to improve its performance. In figures 5.8, 5.9 and 5.10 the boxplots plot the quartiles of the experiments. Figures 5.5, 5.6 and 5.7, show that the GA is able to improve the initial performance. The lines show the GA could have improved the performance if there would be more generations used, but probably not that much. The reason for a low initial performance in comparison with the randomly initialized liquid, is that the weights are randomly initialized in a GA. The results for the a noise rate of 0.01 are good, compared to a Liquid State Machine with random initialized liquid. The trained liquid is clearly an improvement in comparison with the randomly initialized liquid. The liquid is improved by almost twenty one percent. Also only summing the inputs clearly shows a lower performance than using a randomly initialized network, which shows that the liquid is at least capable of mapping the data in such a way that it is better represented then it initially has been. Evolino even does a better job than the trained liquid.

The learning curves with noises of 0.05 and 0.1, show that the LSM implemented here has its shortcomings. This is because in comparison with a noise of 0.01, the performances drop with quite a lot. Although with a noise of 0.05 the performance is well over the performance of a random initialized liquid, the performance decrease is high. Also, in comparison with Evolino, the performance drop is quite large. The decrease between 0.05 and 0.1 is not that high, but as with a noise of 0.05, comparing Evolino with the trained liquid, the difference is large. The performance of Evolino itself shows that Evolino can cope with higher noise. One reason that Evolino outperforms the LSM is that the memory cells are better able to make a pattern, which are than better able to map the time-series pattern in such a way that the performance of the readout network is increased.

The last experiment done using the training algorithms is on the music classification task. The average performances are shown in figure 5.12 marked with a red +. The training curve of the GA can be found in figure 5.11. The quartiles can be found in figure 5.12. The first note to make is that using no recurrent network, works better than Evolino. Also in contrary to the results on the movement set, the results in 5.12 show that the trained LSM can outperform Evolino. The learning curve of the GA trained liquid in figure 5.11 however, shows there is not much to improve from. A reason for this may be that in the music classification set using random initialized weights already boosts the performance without learning. What the GA does here is not really find a better solution for the liquid by using a heuristic, but initialize a large number of liquids and pick the one with the highest performance. A reason for the low performance using a recurrent neural network compared with the summed input, may be the fact that the samples that are used, have little activation. There are 72 inputs each timestep and only a few give an input of one. This means there is not much activation in the network, which makes it difficult for the readout network the separate different patterns. Also a reason that Evolino is performing worse than the other systems, can be found by looking at figure 5.12. The distances between the quartiles is large in comparison with the other systems. Also Evolino is able to get a performance of over 90 percent, but only once. A reason for this may be the settings for Evolino. For example it might be that the number of generations that Evolino is trained is too low. But it could also be that the populationsize is too low, which causes the algorithm to optimize in a bad local optimum. One last reason may be the size of the network. The size is almost the same as with the movement task, but the number of input neurons for the music task is far higher. This is something for future research, because there was no time to further experiment with the parameters.

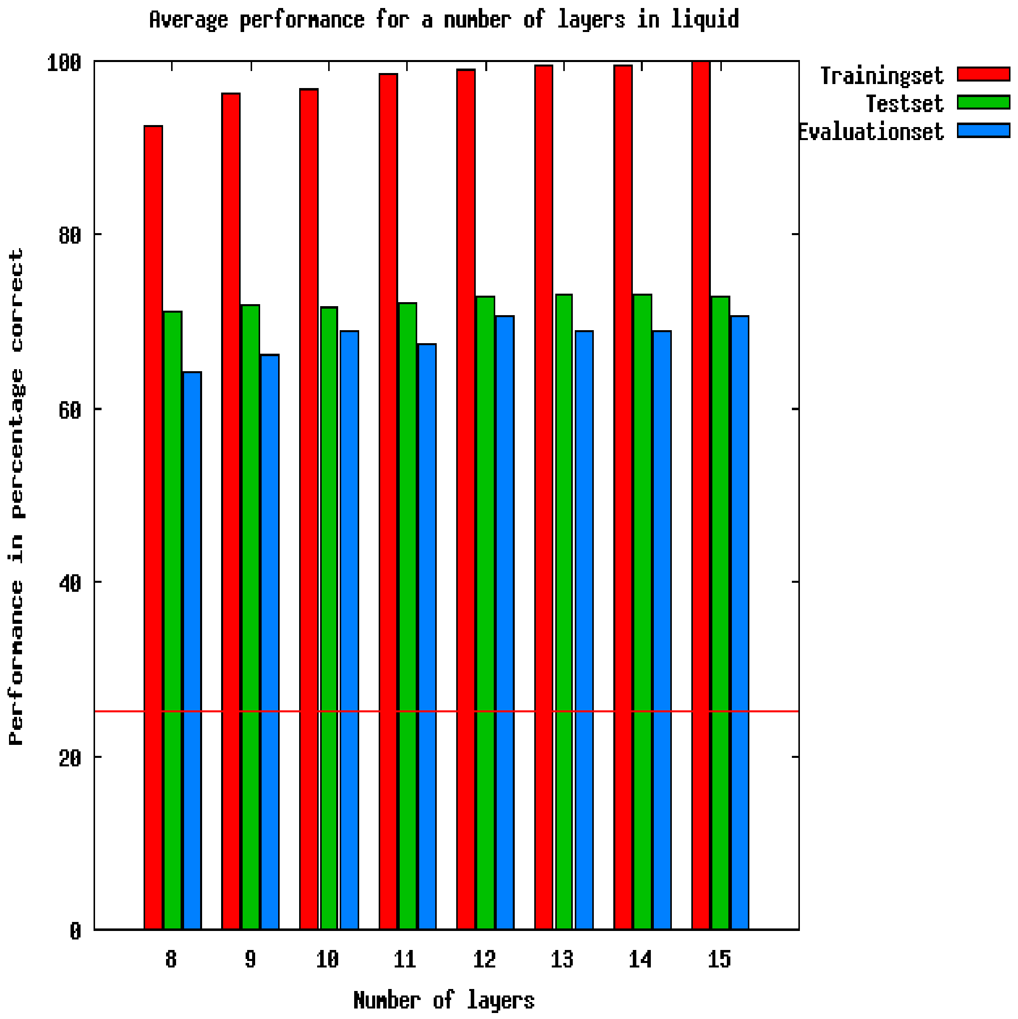


Figure 5.1: Comparison of average performance on movement set with noise of 0.01 with different number of layers.

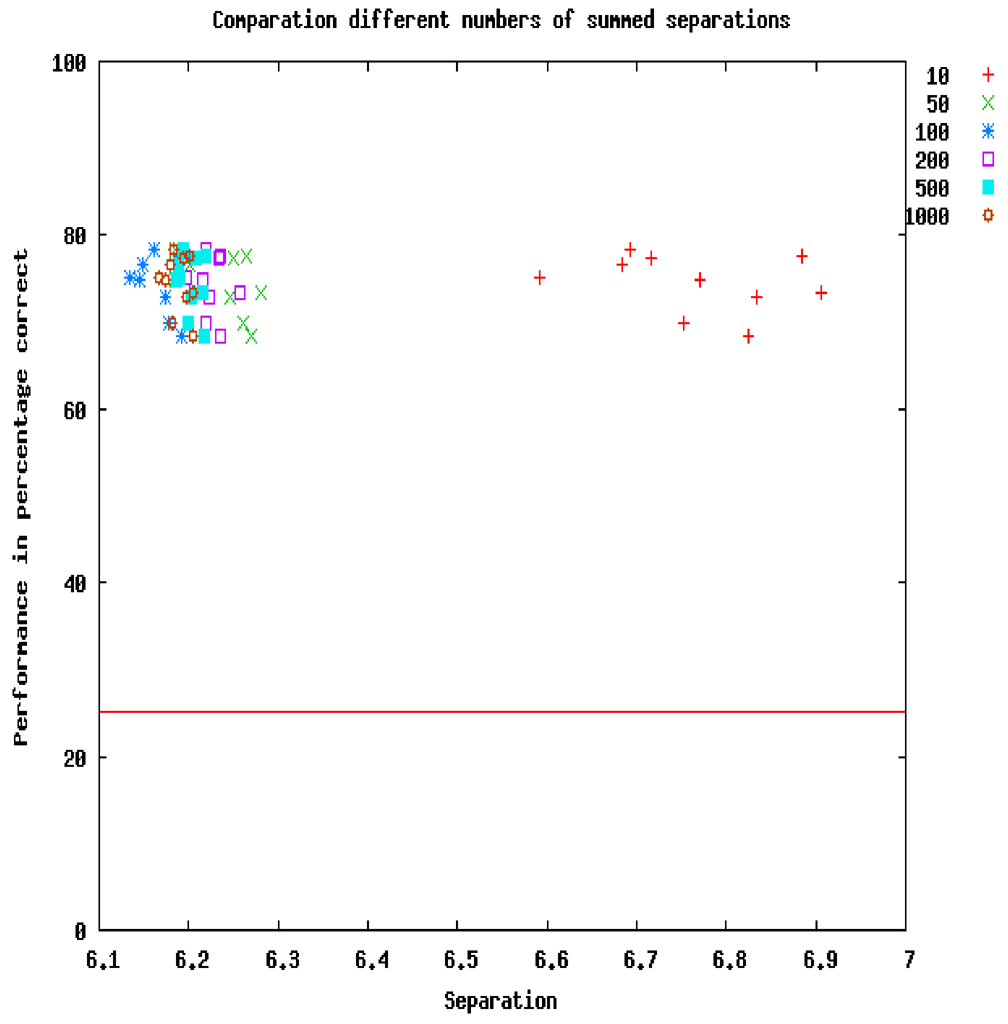


Figure 5.2: Plot of spread, where the separation is compared with the performance of the readout network with different number of sums for the separation function. Note that the x-axis does not start at zero.

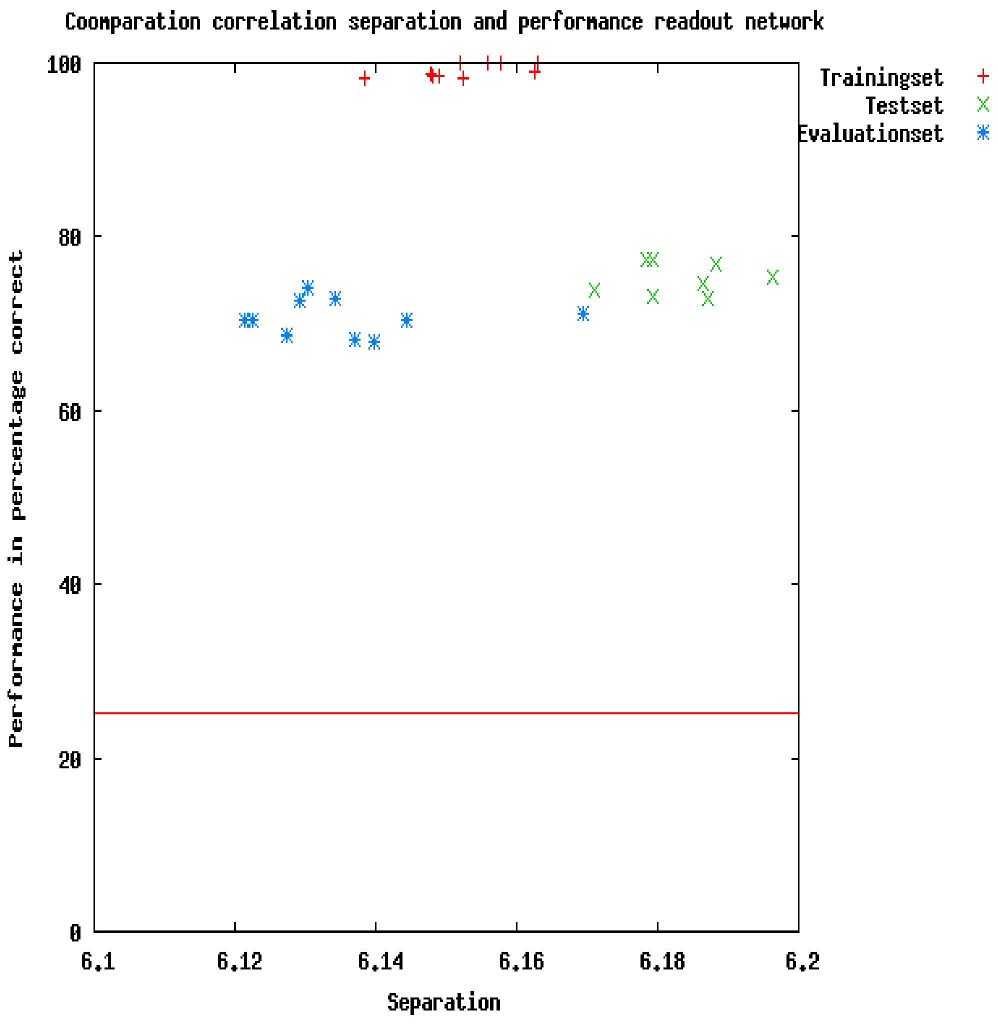


Figure 5.3: Plot of spread, where the separation is compared with the performance of the readout network. Note that the x-axis does not start at zero.

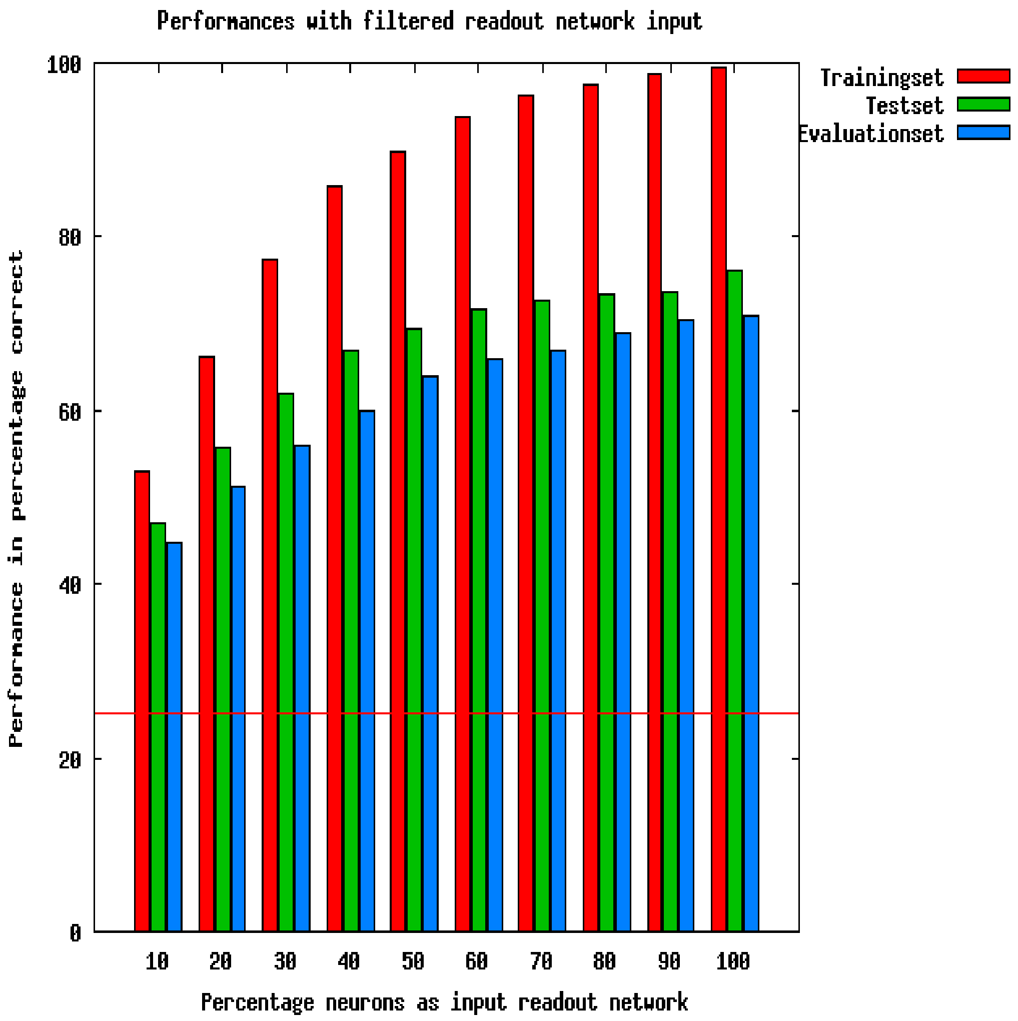


Figure 5.4: Bardiagram which plots the performances with different percentages of neurons used

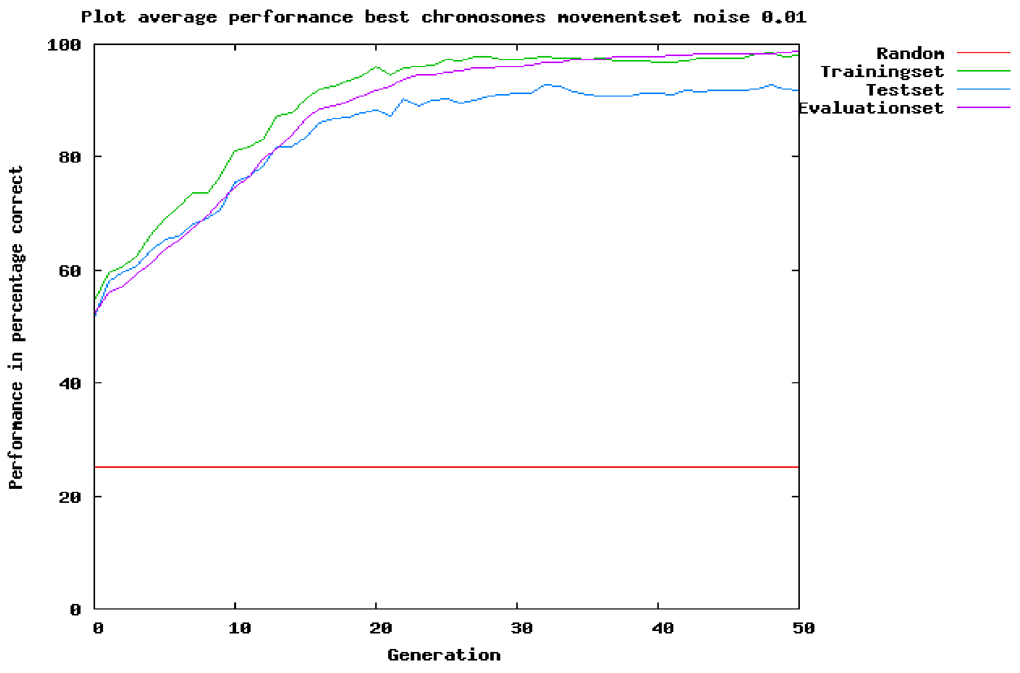


Figure 5.5: Best liquid each generation on the movement classification task with 0.01 chance noise. The random line is the line where the performance is as good as a random generator.

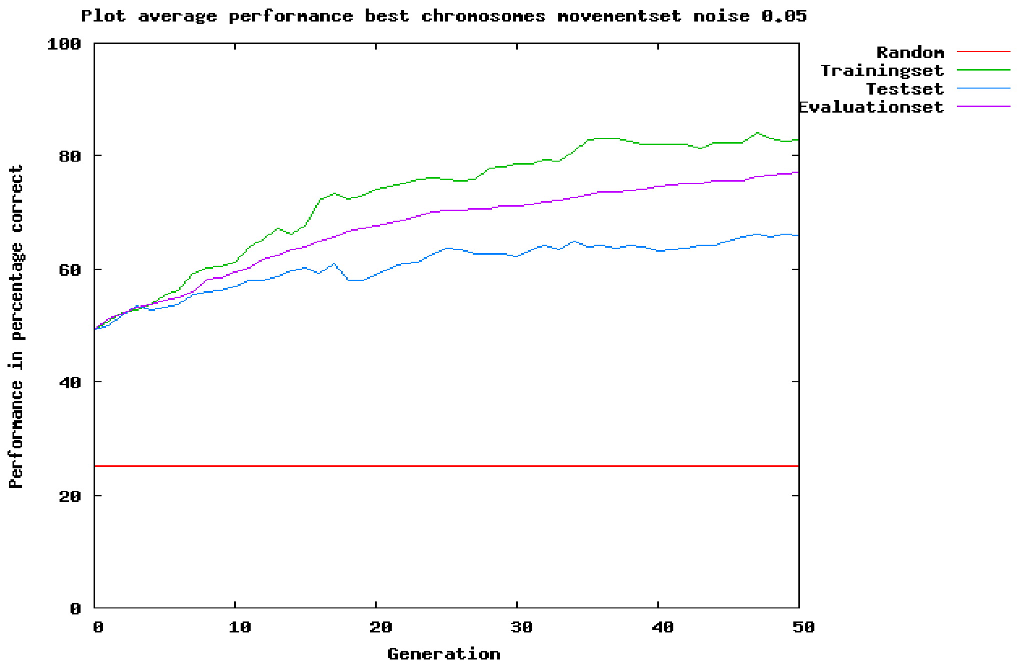


Figure 5.6: Best liquid each generation on the movement classification task with 0.05 chance noise. The random line is the line where the performance is as good as a random generator.

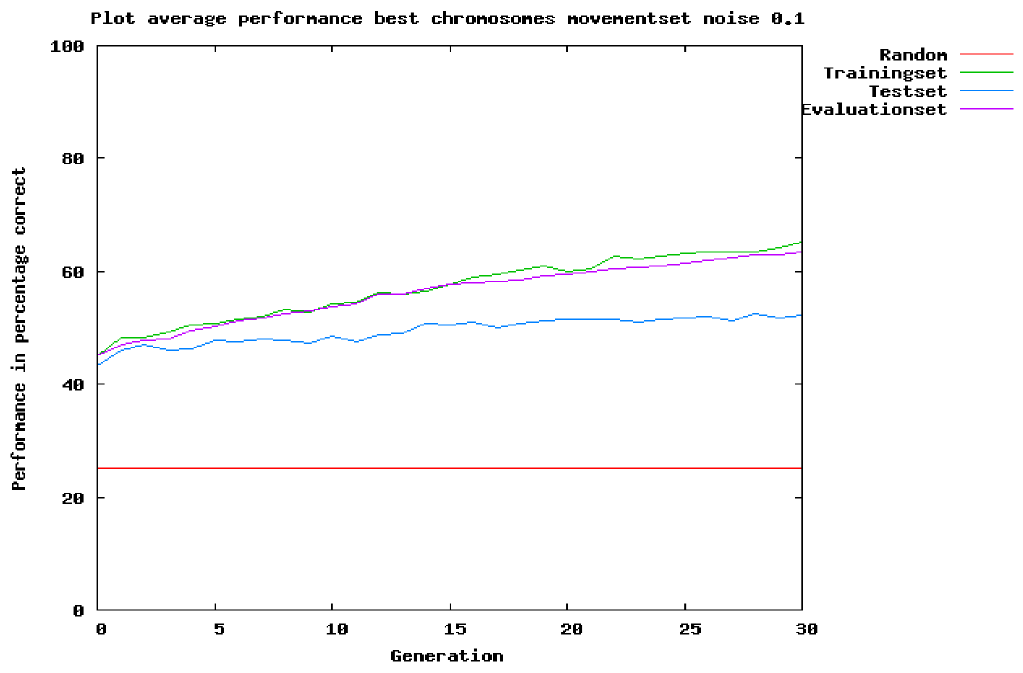


Figure 5.7: Best liquid each generation on the movement classification task with 0.1 chance noise. The random line is the line where the performance is as good as a random generator.

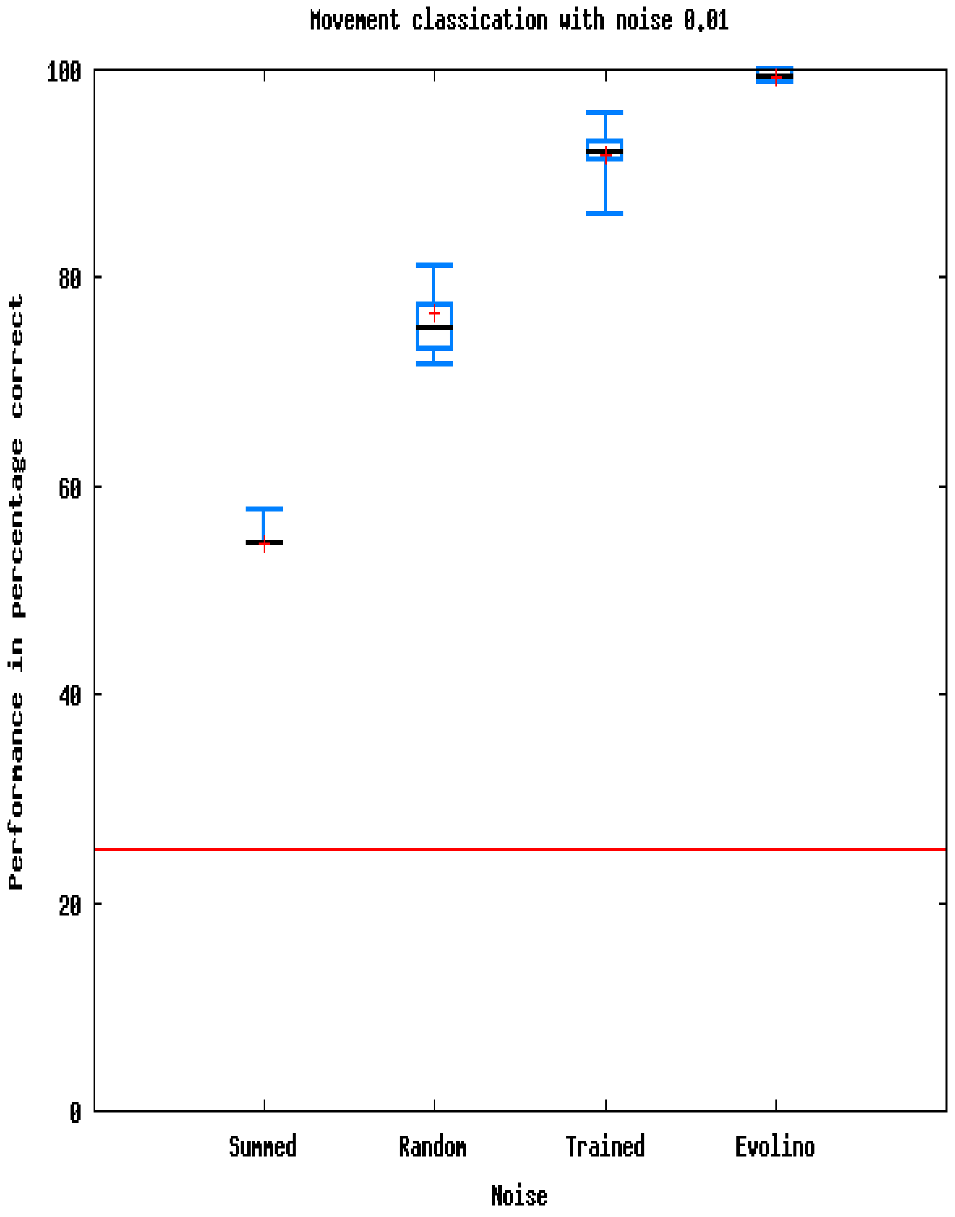


Figure 5.8: Quartiles and average for the movement classification set with a noise of 0.01. Average is the red + in the boxplot. Statistics are from a randomly initialized liquid, a readout network using summed input, a trained liquid with a Genetic Algorithm and Evolino.

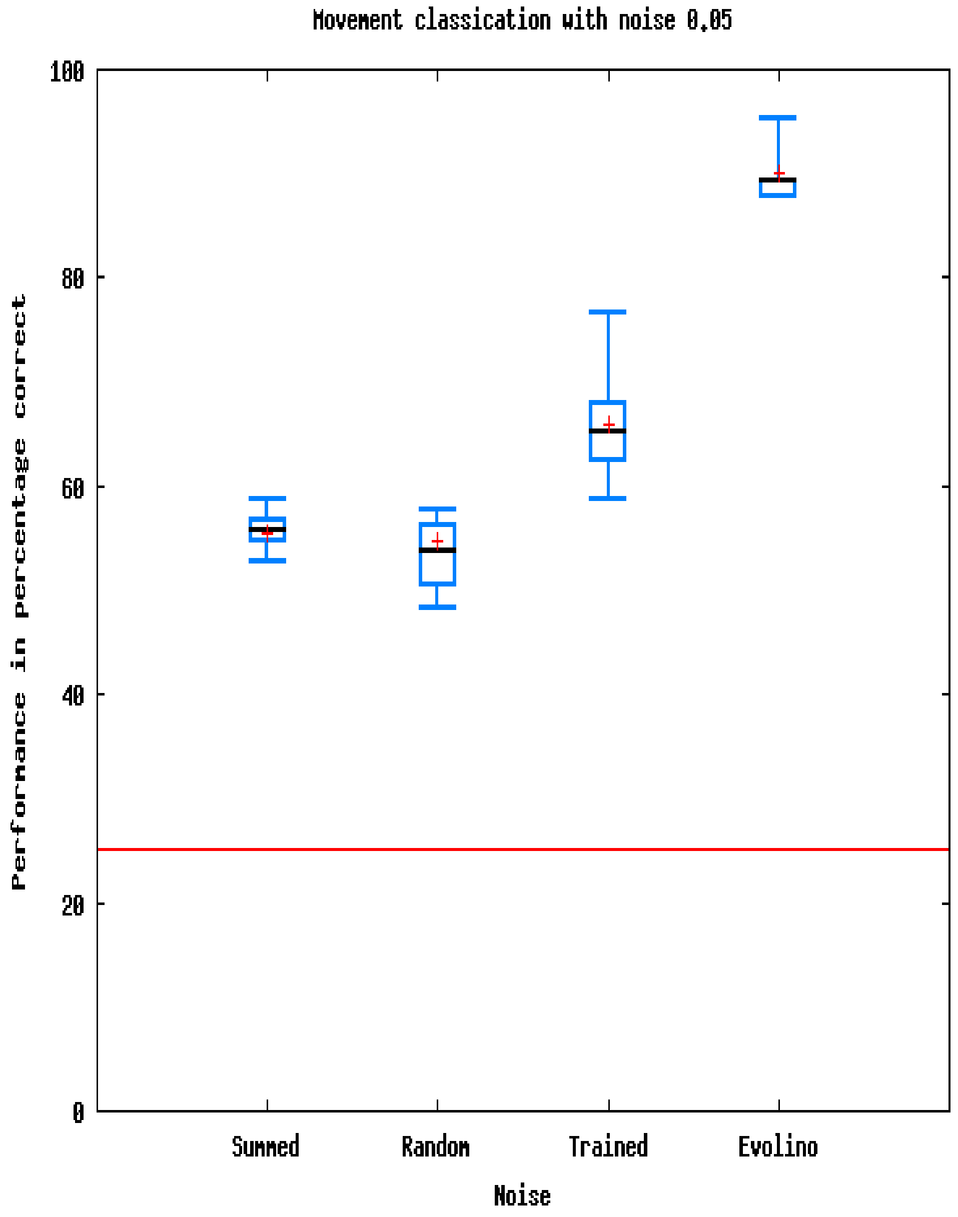


Figure 5.9: Quartiles and average for the movement classification set with a noise of 0.05. Average is the red + in the boxplot. Statistics are from a randomly initialized liquid, a readout network using summed input, a trained liquid with a Genetic Algorithm and Evolino.

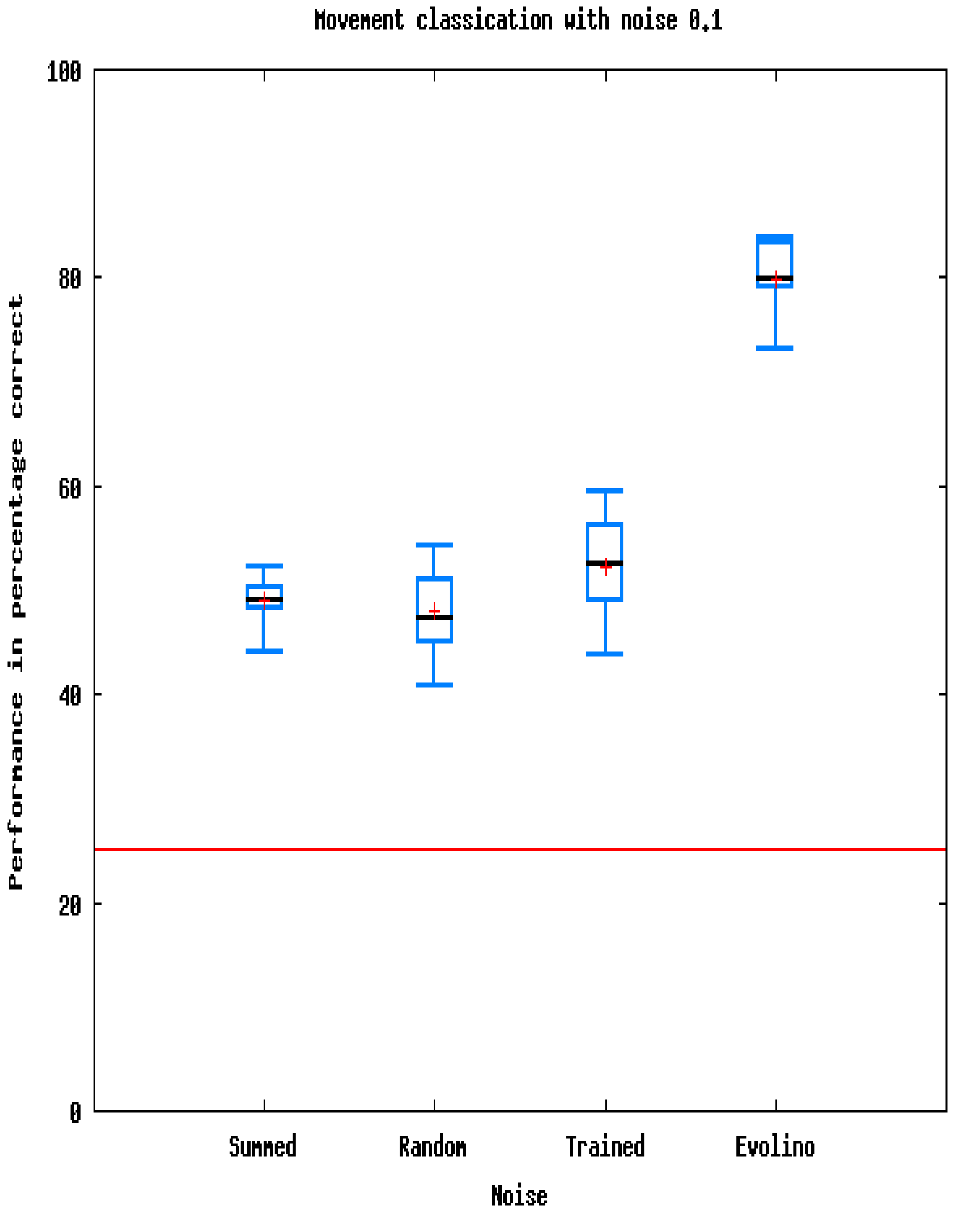


Figure 5.10: Quartiles and average for the movement classification set with a noise of 0.1. Average is the red + in the boxplot. Statistics are from a randomly initialized liquid, a readout network using summed input, a trained liquid with a Genetic Algorithm and Evolino.

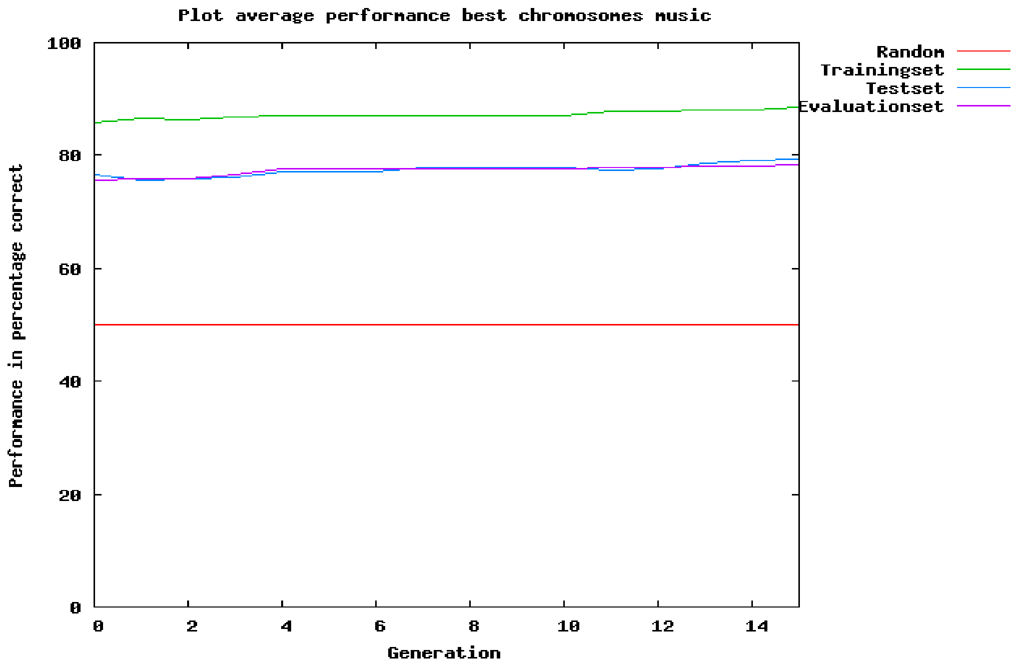


Figure 5.11: Best liquid each generation on the music classification task. The random line is the line where the performance is as good as a random generator.

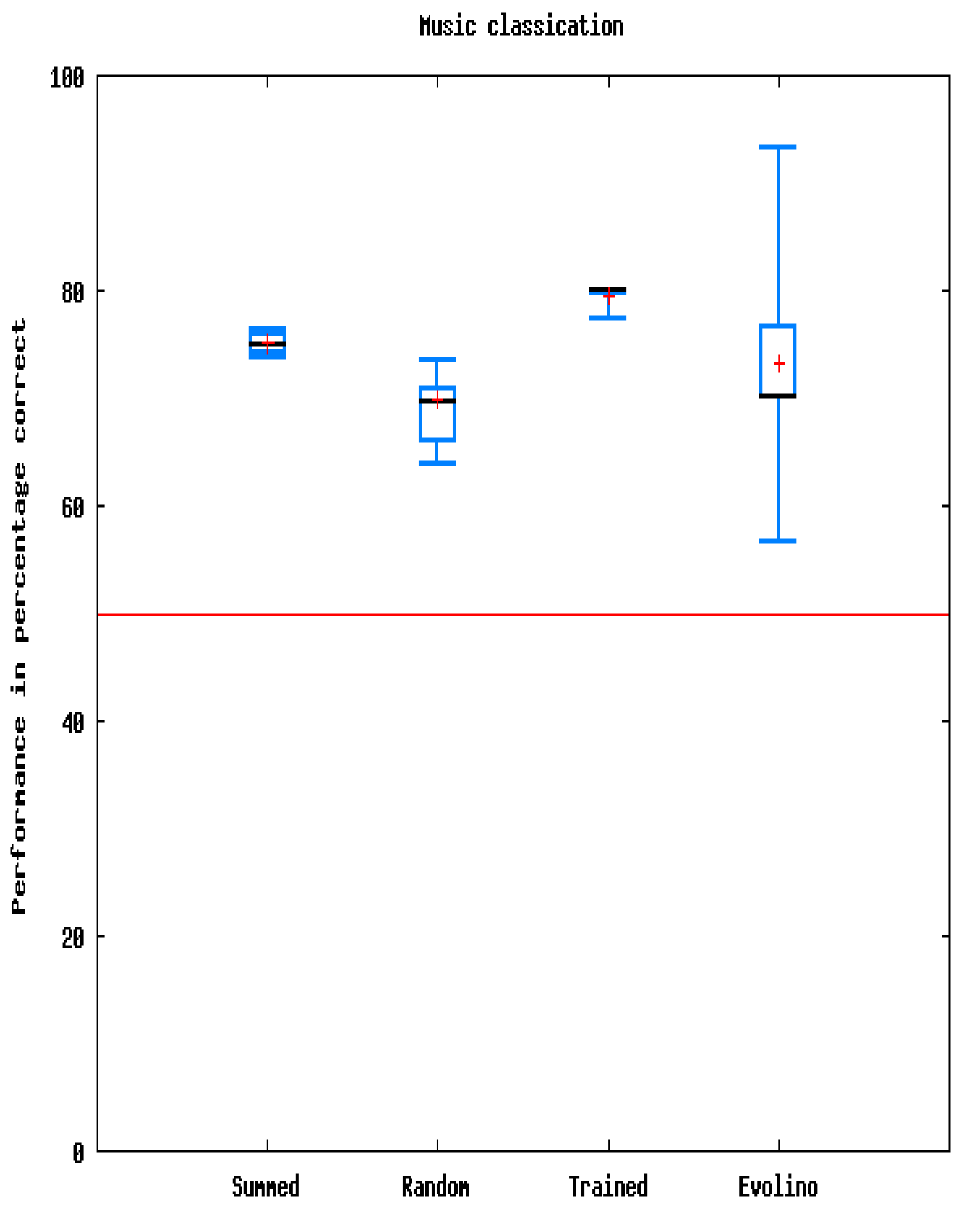


Figure 5.12: Quartiles and average for the music classification set. Average is the red + in the boxplot. Statistics are from a randomly initialized liquid, a readout network using summed input, a trained liquid with a Genetic Algorithm and Evolino.

Chapter 6

Conclusion

6.1 Discussion

One of the research goals was to find a better solution in calculating the performance of a Liquid State Machine. The separation function as used in the experiments, is probably not a good method to use for optimization in the setting that is used here. It could be that this function could have worked for linear systems, but Feed-Forward Neural Networks are capable of solving the problem in a different way. This may thus reduce the correlation between the separation function and the performance of the readout network. Another point to make is that the increase in speed in comparison with calculating the performance using a neural network, is not that significant. Although it is faster than using a neural network, most of the computation time goes to running the liquid. However the separation function only needs one dataset, while in the experiments for example with using the Genetic Algorithm, there are two datasets to be used (one for training the neural network and one for evaluating it, although this is not necessary), thus increasing computation time. But the decrease in computation time only helps if it actually does its job and this does not seem to be true in these settings.

The results of the training paradigms for the Liquid State Machine are not good. First of all the Reinforcement Learning algorithm showed that there is little improvement that can be made on optimizing only the wiring. The Genetic Algorithm performs better, but this is due to the use of different weights. One part of the randomly initialized liquid that is not used in these experiments is the diversity in weights. This is mostly due to the fact that it is difficult to find a good heuristic to initialize the weights and just random initializing weights does not improve the performance (see figures 5.5, 5.6 and 5.7 initial performance). This is thus a part that could be improved using an optimization algorithm. In comparison with Reinforcement Learning, it is easy to implement a Genetic Algorithm that can cope with the real valued weights. This is one disadvantage of using a RL algorithm, because it is difficult to find a method to learn real valued numbers. GA's solve this by using mutation which can possibly change the weights into any other real number. Searching for better weights, makes the search space bigger and possibly increases the computation time, but there are already enough examples of Genetic Algorithms that optimize problems that use real numbers [24].

The results also showed that Evolino can handle the movement task well. With the movement classification set, Evolino simply outperforms the LSM. Although with a noise of 0.01 the difference is not that high, with higher noise rates, Evolino shows that it simply is a better system than the implementation of the LSM that is used in the experiments. Although the results showed that the LSM at least outperforms an FNN that does not use a recurrent network for its input. The main reason for the bad performance in comparison with Evolino is that the Spiking Neural Network that is implemented here, is not sufficiently capable of mapping the time-series data in a way that can be used for the readout network. The Long Short-Term Memory network does this job far better. Also the readout moments of the Spiking Neural Network consisted mostly of spikes (summed ones), while the Long Short-Term Memory network uses real numbers as output. Although spikes can give a sufficient amount of information, for a readout network it may be too little. The LSTM has more different output signals in comparison with the two of the Spiking Neural Network (namely spiking or not spiking). One other thing to note is that a part of the information of Spiking Networks comes from the spike timings. By adding spikes up, this timing is thrown away. Also another advantage of the memory cell is that there is more control over the signal in the memory cell. The memory cell can control what part of an incoming signal should be used in the core of the cell and which part does not. Also it can decide if there should be any output and how much of the signal should be outputted. This makes a big difference with the spiking neuron, although another form of this system is implemented in the spiking neuron, by being able to let activation leak away or stopping signals to be outputted by having a theshold. The difference is that the memory cell has far more control and is more precise, by having weights between the output gate of the sending cell and all the gates of the receiving cell. Another disadvantage of the Spiking Neural Network that is used here, is that the computation time it takes for the Liquid State Machine to be trained is higher than that of Evolino. One reason may be the way the systems are programmed. But the main reason is probably that the Long Short-Term Memory network needs far less units to get a good performance in comparison with the Spiking Neural Network that is used. While on the movement classification set, the Spiking Neural Network uses 108 neurons, the Long Short-Term Memory only uses 20 memory cells. Although the memory cells are connected with each other by four different connections, it still needs less computation time for the network to map input pattern in a good way for the readout network. Taking the movement task, for the Spiking Network, there are possibly $9 \times 108 + 108 \times 108 = 12636$ connections. With LSTM this is $9 \times 20 \times 4 + 20 \times 20 \times 4 = 2320$ connections. This is also about the number of computations that should be made each timestep, thus making the computation time of the Spiking Network, more than five times larger than with LSTM. Next to that, Evolutionary Algorithms are computational heavy algorithms, each generation, a high number of fitnesses need to be calculated. This leads to an increae in computation time when comparing the Spiking Network with LSTM.

One other reason that Evolino performs better than the LSM, could be the type of training algorithm. Although this is probably not that influential as the structure of the recurrent networks, it may have some influence. GA's are a more straightforward method in solving the problem and do the job well. ESP could be a bit less stable, as it is a co-evolutionary system, which means that fitnesses of individual chromosomes are dependent on each other. But it learns to specialize each memory cell in the network, which may improve the performance. This specialization does not necessarily take place with a normal GA. Here, the optimization is not done for single neurons but on complete networks. It could thus be that by optimizing single neurons, better solutions can be found than when using a normal GA.

With the music classification task, both LSM and Evolino do not perform that well compared to a simple summation over the input for a Feed-Forward Neural Network. There are some reasons for this, one is the lack of time, which made it difficult to find good parameters and also forced to use lower population sizes, less generations to optimize and smaller network sizes. One reason for this is that optimizing the music task is far more computationally intensive than with the movement task, because there are much longer samples in the dataset. This caused difficulties in finding optimal settings and with the time-constraints taken into account, the settings should take up as little computation time as possible. With Evolino there is a high variation in the different runs. The reason for this may be that the ESP algorithm is not yet converged to a optimal solution. But it could also be that the subpopulations are too small. If in an Evolutionary Algorithm, the population size is too small, the population will likely get stuck in a local optimum, which does not give a good performance, this is due to the low diversity in the gene-pool. Although mutation may solve this problem, mutation is not designed to change genestrings drastically and thus jump out of local maxima, because mutation is also a local search method. Having crossover and a big population will give more diversity, but it will also take a lot longer to optimize the population. There is thus a trade-off to be made in how big the population should be, how long an Evolutionary Algorithm should be trained and how much time it will take. Also it is important to see how much of an increase in performance it will get, because it is not always worth the time to increase the performance a couple of tenths of percents if it takes up days or months to find these better solutions. Also notable is the lack of increase in performance with the GA, when comparing the first generation to the last. One reason may be that the randomly initialized weights already are sufficient enough to separate the output patterns, instead of learning the weights.

Another reason for the relatively bad performance, in comparison with the other systems, is the lack of activity in the input pattern. This makes it difficult certainly for a Spiking Neural Network to get enough activation for the readout network to divide two readout patterns. An SNN should have enough activation, otherwise neurons will not spike and thus cannot create different patterns. Also a reason that Evolino has such a low performance is that it uses quite a lot less data (it does not use the evaluation set). With the dataset already limited, it could be hard for Evolino to find all the different patterns in the input. This is also a problem for every system, the dataset is already limited. The dataset is split up into three parts, with only a few samples. This increases the chances of overfitting as it often does not find all the properties that are in the dataset to learn. A point to make is that although the same dataset is used in [16], the data used here is split up into three parts, with a test, evaluation and training set, while in [16] the set was split up in two parts, namely a training and test set.

6.2 Future Research

The results showed that the liquid used in the experiments has its limitations. But also the training algorithms may have something to do with this. In this section a number of different ideas are given to look further into the problem.

One part to look further into, is the implementation of the liquid. For example a part to do further research on is the implementation of the neurons. Maybe a different type of spike neuron may improve the performance. For example the memory cell had more control over the signal, this is something which could be implemented in the spiking neuron as well. But another part to look at is how well readout networks can cope with the limited information that spiking neurons send out. Because they only have two different signals, the information is limited in comparison with continuous recurrent networks. A comparison between different continuous and Spiking Neural Networks, showing what the differences are between these two types of networks, gives a better view on the abilities of the two different types of networks. Also the methods of reading out the states is important certainly for Spiking Networks. These networks carry a lot of information by the timing of spikes. Other methods to keep the timing in the input for the readout network, could increase the performance and it easier for the readout network to differentiate the readout patterns.

The second part which could be further research, are the different optimization algorithms. Although some of the more important and well known optimization algorithms in Machine Learning are used, there may be other algorithms that could be used. One is to compare different implementations of an Evolutionary Algorithm. For example Evolino uses ESP, for a good comparison the ESP should also be implemented using the LSM, this may show how much the ESP is beneficial to learning. Also another part is to use a Reinforcement Learning algorithm that can also optimize the weights. Furthermore other types of learning methods could be used. For example to combine an Evolutionary Algorithm with a Gradient Descent method. Maybe if these two methods are combined, it can increase the performance by using the Evolutionary Algorithm to find a 'raw' solution and optimize it with a Gradient Descent method. This may decrease the time that the Gradient Descent method needs to learn and can thus cope with some of its problems with temporal dependencies a little better. Furthermore, there are more ways to optimize an algorith. For example, optimizing the parameters, like the λ, instead of a complete wiring, may already increase the performance quite a bit, this has already been done in [9], but here Echo State Networks [10] are used. The question is if the increase in performance of a liquid that has an optimized wiring and weights setting performs that much better than a liquid with only an optimized parameter setting. Also the question is if the computation time that it takes to optimize the wiring and weights is worth spending instead of having a little less performing liquid, but which takes up less computation time to optimize.

The implementation of the program may be important for lowering the computation time. Genetic Algorithms, certainly with optimization of LSMs, are a computationally heavy task. To solve this, programs can be run on a cluster, by distributing work over multiple processors. Genetic Algorithms are ideal for this implementation, because a Genetic Algorithm consists of a population where calculating the fitness of one chromosome is not dependent on the fitness of another. Thus the fitness calculation for each chromosome can be done parallel on multiple processors. This could possibly decrease the time it takes for one run to finish. This will than make it more easy to search for optimal parameter settings and do more experiments. Due to time-constraints it was not possible to implement this for the experiments in this thesis.

The separation function in the end gave disappointing results. But there are more methods to implement the separation. Also, there may be other methods to compute the performance, that do not take up as much as computation time as it does now.

One goal of AI algorithms is to outperform humans in a task. Here to outperform a human means that the algorithm should be able to classify more samples correctly. This comparison is not made in this thesis but is certainly possible. The movement classification task is a task that humans are capable to do. Certainly with high noise rates, the performance of humans in doing this job, can be compared with for example Evolino. It gives an insight in how well Evolino performs, but also if there are implementations that are getting close, or are better than humans for this task. Also other datasets could be used to compare the performance of Evolino and LSMs with humans.

Appendix A

Solution for Calculating Initial Q-values

To find the correct Q-values, it first has to be done for an agent with two states $Q_t(a)$, the default state and $Q_t(b)$ the state that should be calculated:

$$e^{-\frac{dist(n_i,n_j)}{\lambda}} = \frac{e^{\frac{Q_t(a)}{\tau}}}{e^{\frac{Q_t(a)}{\tau}} + e^{\frac{Q_t(b)}{\tau}}}. \tag{A.1}$$

Here $dist_a(n_i, n_j)$ is the distance between neurons n_i and n_j. Equation A.1 is a bit counterintuitive as it would be more intuitive to calculate the chance for $Q_t(b)$, but this would give problems further up in the solution. $e^{-\frac{dist(n_i,n_j)}{\lambda}}$ can also be found in section 2.4.2. In further equations, to decrease the complexity of the equation, $e^{-\frac{dist(n_i,n_j)}{\lambda}}$ will be swapped with p_a and $e^{\frac{Q_t(s)}{\tau}}$ will be swapped for the constant c as this value is already known.

$$p_a = \frac{c}{c + e^{\frac{Q_t(b)}{\tau}}}. \tag{A.2}$$

Next the fraction is cleared by multiplying by $c + e^{\frac{Q_t(b)}{\tau}}$:

$$p_a(c + e^{\frac{Q_t(b)}{\tau}}) = c. \tag{A.3}$$

After this, $c + e^{\frac{Q_t(b)}{\tau}}$ is multiplied by p_a resulting in:

$$p_a c + p_a e^{\frac{Q_t(b)}{\tau}} = c. \tag{A.4}$$

Then $p_a c$ is subtracted:

$$p_a e^{\frac{Q_t(b)}{\tau}} = c - p_a c. \tag{A.5}$$

Then the equation is divided by p_a:

$$e^{\frac{Q_t(b)}{\tau}} = \frac{c}{p_a} - c, \tag{A.6}$$

and then the natural logarithm is taken to remove the exponent $e^{\frac{Q_t(b)}{\tau}}$:

$$\frac{Q_t(b)}{\tau} = \ln(\frac{c}{p_a} - c), \tag{A.7}$$

and finally to remove the τ from the left side, the equation is multiplied by τ:

$$Q_t(b) = \ln(\frac{c}{p_a} - c) \cdot \tau. \tag{A.8}$$

Replacing c and p_a this will become:

$$Q_t(b) = \ln(\frac{e^{\frac{Q_t(a)}{\tau}}}{e^{-\frac{dist(n_i,n_j)}{\lambda}}} - e^{\frac{Q_t(a)}{\tau}}) \cdot \tau. \tag{A.9}$$

The problem however is that sometimes an agent has more than two states. To solve this problem, the probabilities are recalculated to only be in a relation with the probability of the default Q-value:

$$p_b^{new} = \frac{p_b}{p_b + p_a}. \tag{A.10}$$

To make it more clear here is an example. Suppose there are three states $\{a, b, c\}$. The probabilities for these states are: p_a is 0.2, p_b is 0.5 and p_c is 0.3. Furthermore $Q_0(a)$ is set to four. Also to keep it simple, both λ and τ are set to one. First $Q_0(b)$ is calculated (here $Q_0(b)$ is the Q-value of state b at time 0). To do this, first equation A.10 is used to get p_a^{new} related to p_b:

$$\frac{0.2}{0.5 + 0.2} = 0.286. \tag{A.11}$$

Now to find $Q_t(b)$, equation A.9 is filled in:

$$\ln(\frac{e^4}{0.286} - e^4) = 4.915, \tag{A.12}$$

and for $Q_0(c)$ this will be:

$$\frac{0.2}{0.3+0.2} = 0.4. \tag{A.13}$$

This is used in equation A.9:

$$\ln(\frac{e^4}{0.4} - e^4) = 4.405. \tag{A.14}$$

Now to check if this is true first calculate p_a:

$$\frac{e^4}{e^4 + e^{4.915} + e^{4.405}} = 0.2, \tag{A.15}$$

and for p_b this is:

$$\frac{e^{4.915}}{e^4 + e^{4.915} + e^{4.405}} = 0.5, \tag{A.16}$$

and finally for p_c:

$$\frac{e^{4.405}}{e^4 + e^{4.915} + e^{4.405}} = 0.3. \tag{A.17}$$

Bibliography

[1] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994.

[2] M. Biehl and H. Schwarze. Learning by online gradient descent. *Journal of Physics A: Mathematical and General*, 28(3):643–656, 1995.

[3] S.M. Bohte, J.N. Kok, and H. La Poutré. Spike-prop: error-backprogation in multi-layer networks of spiking neurons. *Neural Computation*, 48:17–37, 2002.

[4] D.A. Coley. *An introduction to genetic algorithms for scientists and engineers.* World Scientific, 1999.

[5] J.L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, April-June 1990.

[6] D.O. Hebb. *The organization of behavior: A Neuralpsychological Theory.* Wiley, 1949.

[7] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 6(2):107–116, 1998.

[8] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[9] K. Ishii, T. van der Zant, V. Becanovic, and P. Plöger. Identification of motion with echo state network. *Conference: OCEANS/TECHNO-OCEAN (OTO) ¡2004, Kobe¿*, 2004.

[10] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. Gmd report 148, German National Research Center for Information Technology, 2001.

[11] K. J. Lang, A. H. Waibel, and G. E. Hinton. A time-delay neural network architecture for isolated word recognition. *Neural Netw.*, 3(1):23–43, 1990.

[12] T. Lin, B. G. Horne, P. Tiño, and C. L. Giles. Learning long-term dependencies is not as difficult with NARX networks. In *Advances in Neural Information Processing Systems*, volume 8, pages 577–583. The MIT Press, 1996.

[13] Mitchell T. M. *Machine Learning.* The McGraw-Hill Companies, 1997.

[14] 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.

[15] D. E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(1-3):11–32, 1996.

[16] L. Pape, de Gruijl J., and M. Wiering. Democratic liquid state machines for music classification. *Speech, Audio, Image and Biomedical Signal Processing using Neural Networks, Bookseries: Studies in Computational Intelligence*, 83, 2008.

[17] B. A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228, September 1995.

[18] R. Penrose. A generalized inverse for matrices. *Proceedings of the Cambridge Philosophy Society*, 51:406–413, 1955.

[19] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[20] A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical report, Cambridge University Engineering Department, 1987.

[21] F Rosenblatt. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms.* Spartan Books, 1962.

[22] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. pages 696–699, 1988.

[23] J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. Gomez. Training recurrent networks by evolino. *Neural Computation*, 19(3):757–779, 2007.

[24] R.S. Sexton, R.E. Dorsey, and J.D. Johnson. Optimization of neural networks: A comparative analysis of the genetic algorithm and simulated annealing. *European Journal of Operational Research*, 114(3):589–601, May 1999.

[25] S. R. Sutton and A. G. Barto. *Reinforcement Learning An Introduction.* MIT Press, 1998.

[26] J. Vreeken. Spiking neural networks, an introduction. Technical report, Institute for Information and Computing Sciences, Utrecht University, 2003.

[27] P. J Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, 1974.

[28] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin and D. E. Rumelhart, editors, *Back-propagation: Theory, Architectures and Applications*, pages 433–486. Lawrence Erlbaum Publishers, Hillsdale, N.J., 1995.