University of Plymouth

PEARL **https://pearl.plymouth.ac.uk**

04 University of Plymouth Research Theses 01 Research Theses Main Collection

2023

Learning and planning for autonomous systems with emergent hierarchical representations and decaying short-term memory

Bogdan, Piotr Aleksander

http://hdl.handle.net/10026.1/20533

http://dx.doi.org/10.24382/dspace/356
University of Plymouth

All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.

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

UNIVERSITY OF PLYMOUTH

Learning and planning for autonomous systems with emergent hierarchical representations and decaying short-term memory

by

Piotr Aleksander Bogdan

A thesis submitted to the University of Plymouth
in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

School of Engineering, Computing and Mathematics

March 2023

Acknowledgements

I would like to thank Dr Torbjørn Dahl the original Director of Studies for being an inspiration for me and this research project. This endeavour would not have happen without his patience and dedication. The extend of his scientific and mental support expressed towards me cannot be overstated. I wish him the very best in his future career.

I would also like to thank Dr Thomas Wennekers who was initially my second supervisor but in the last year stepped up to the position of Director of Studies. His patience and understanding carried me through the difficult process of writing up.

I would like to thank the whole body of staff and students of the University of Plymouth for creating an environment where ideas and understanding of surrounding world can develop.

I would also like to thank my friends and family for providing emotional support and occasional distraction.

Author's declaration

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award without prior agreement of the Doctoral College Quality Sub-Committee.

This thesis has been proofread by a third party; no factual changes or additions or amendments to the argument were made as a result of this process. A copy of the thesis prior to proofreading will be made available to the examiners upon request.

Work submitted for this research degree at the University of Plymouth has not formed part of any other degree either at the University of Plymouth or at another establishment.

Word count of main body of thesis: **20044**

[illegible]

Signed: ______________________

Date: 2nd March 2023

Abstract

Piotr Aleksander Bogdan

Learning and planning for autonomous systems with emergent hierarchical representations and decaying short-term memory

How do we create machines with the ability to capture, record and recall memories of past experience? How should these machines choose the most optimal action based on those stored memories? These seem like crucial questions for creating intelligent machines capable of learning from experience. The field of Artificial Intelligence (AI) is trying to reproduce such capabilities with increasing success. Currently a large portion of AI algorithms are focusing on making decisions based on big sets of learned past experience examples in the form of instantaneous input-output mapping. They operate as discrete models where time is collapsed into independent signal samples. Yet the dimension of time is the most fundamental source of perception, and the presence and absence of the signal is only visible by its changes in time. This work combines features of established neural network algorithms to create a new approach to the processing of temporal signals. Considering either agent-environment division present in reinforcement learning (RL) or controller-process division in control theory, there is always the intelligent part, agent or controller, which tries to control the passive, mostly deterministic part, environment or process. As the complexity of the problem grows, the coupling between the controller and the controlled part starts to become more physically limited. With limited perception, the controller has to resolve to building an abstract model of the controlled process or environment in order to be able to take fully informed actions. Presented in this thesis is a new artificial neural network capable of creating an unsupervised temporal model of the signal that can then be used as an abstract environment model for the controller. The network is structured as multilayer hierarchical composition of self-organising maps, augmented by short term memory in the form of wave-delay lines. Each layer performs temporal signal decomposition with a progressively larger time spectrum. The research analyses the network performance in creating abstract signal models on a range of synthetic and real world signals. It then introduces simple reinforcement learning additions, that allow the network to solve simple toy RL benchmarks.

Contents

List of Figures

Introduction

Artificial agents operating in the real world commonly have limited access to sense the surrounding environment. Usually due to a physical inability to sense certain environmental features. In such situations, where some information is not immediately available through sensory readings, it may still be possible for an agent to achieve a more comprehensive understanding of its situation by considering the overall temporal stream of sensor data, rather than individual sensor readings. These kinds of problems are categorised as partially observable Markov decision processes (POMDP). The POMDP problem formalization has been successfully applied to a number of real world applications Cassandra (1998) through the framework of reinforcement learning (RL). One of the main problems in applying RL techniques to POMDPs is long temporal information lags, i.e., a long delay between the agent receiving a piece of information and the time the agent has to act on that information. Such delays present a specific challenge in that the POMDP algorithm must be able to identify and remember the relevant historical information in order to produce accurate predictions. This problem is formally referred to as "hidden state identification", and it is one of the crucial elements in the credit assignment problem, i.e., the problem of identifying which actions contribute positively to the agent's performance by increasing its expected reward. An agent's estimate of the current POMDP state is called a belief state, sometimes represented as a probability distribution over all possible states. Solving POMDP problems implies calculating a current belief state from a stream of recent observations and actions and using it to select the next action in a way that maximises the expected discounted future reward. This generic approach suffers from two important problems: the curse of dimensionality, caused by the combinatorial explosion of the size of the state space with an increasing number of state parameters, and the curse of history Pineau et al. (2006) whereby the size of the potential state-space also grows exponentially with the number of steps back in time that need to be considered in order to solve a given problem. One way of identifying hidden states is by using feed-forward neural networks with delay-lines Waibel et al. (1989). Delay-line based networks capture the temporal dynamics of a signal by looking at both the current input and a memory of a fixed number of past steps, at the same time. This fixed horizon is a limiting factor in that it cannot capture temporal dynamics that are longer than the number of past steps in the memory. Delay-line algorithms also consider all past steps in memory at every time step. The exponential complexity of this approach, due to the "curse of history", means it is impractical for longer time horizons. As an alternative to delay-lines, algorithms such as value iteration, and its effective implementation point-based value iteration (PBVI) Pineau et al. (2006), instead maintain a running state estimate or belief state. PBVI algorithms maintain an explicit belief-state representation in the form of a probability distribution over all states, which is updated at each step of the algorithm according to transition and observation probabilities. On the other hand, instance-based RL algorithms such as the Nearest Sequence Memory algorithm McCallum (1996) re-calculate an implicit state estimate from scratch at each step of the algorithm, e.g., in the form of the K observation-action sequences in long-term memory that most closely match the sequence currently in short-term memory. In between these two extreme approaches to state estimation, there is a class of algorithms that use implicit state estimations through recurrent neural networks. It is possible to hypothesise an algorithm that borrows from all of those approaches. One that uses short term memory in form of specially modified delay lines with an unsupervised quantization algorithm to smoothly develop a subset of nearest sequence dynamics of the problem. Multiple layers would allow such a network to encapsulate a longer time spectrum and thus develop explicit belief state estimations.

Proposed solution

This thesis proposes a solutions to the hypothesized algorithm in the form of new unsupervised multi-layer feed forward neural network architecture based on Kohonen's self-organizing maps. This algorithm would learn compressed representations of data with arbitrary temporal dynamics. The network architecture is inspired by the Constructivist Learning Architecture presented by Cohen et al. (2002) which successfully modelled empirical data on the development of infant understanding of causal events. Pierris & Dahl (2017) demonstrated how a similar network is able to encode and reproduce robot motions in a POMDP context. The algorithm presented in this thesis goes beyond the work by Pierris in that it learns a compressed representation of the given data, rather than producing lossless encoding. The new algorithm has the potential to overcome the vanishing gradient problem because it relies on a hierarchical structure to encode sequential data, rather than recurrent connections. This thesis presents results showing that the new algorithm can learn representations of data with a range of temporal dynamics and use these representations for sequence prediction. Later the research combined the network with the action selection mechanism in order to produce a novel complete RL algorithm.

The unsupervised network presented can encode signals generated by partially observable Markov decision processes (POMDPs) using multi-layer Kohonen self-organising maps (SOMs) with a layerwise, dynamic short-term memory mechanism in the form of variable-speed wave-delay lines. The ability to model POMDPs is crucial in a variety of problems, including natural language processing, predictive maintenance, autonomous robot control and network routing. The network model improves on existing non-recurrent sequence encoding networks, such as Temporal Convolutional Networks, by removing the requirement to keep large sequence segments in short-term memory. Instead, the network presented here uses fixed-length short-term memory units spread across the levels of the network, providing a hierarchy of increasingly long, but increasingly coarse records. This reduces the memory requirements, as a linear increase in the network's size provides an exponential increase in its capacity. In other non-recurrent models such as temporal convolutional networks, the memory requirement grows linearly with encoding capacity. This research evaluates the effectiveness and robustness of the network architecture through a set of experiments. First, via a simple density-encoded sinusoidal curve with a period of 50 steps, to illustrate its encoding and decomposition mechanism. Second, with a set of signals with different noise levels, and a signal with a temporal distortion, i.e., jitter. The results show that the network can detect signals buried in noise and can extract an underlying signal from noisy data. The network's representation capabilities were tested by training it on the sequential-MNIST dataset and then training a classification algorithm using the network as its input. The results show that the new architecture is effective sequence encoder and that it perform generalisation better than alternative models in the presence of certain types of noise. Finally the network was challenged with encoding the dynamics of a simple toy reinforcement learning benchmark and with an additional state estimation algorithm used to estimate hidden state and solve the problem.

Chapter 1

Literature review

The algorithm proposed here does not stand alone in the diverse field of artificial neural networks. On the contrary it borrows from array of different algorithm. By knowing the context in which it exists and the characteristics that relates to the other we can try to infer its own features and shortcomings. Thus, the literature review starts with an introduction to the scope and general history of neural networks, followed by a description of three general areas from which the new network inherits its characteristics, namely: Self-organizing Maps, Spatio-temporal encoding and Hierarchical Reinforcement Learning.

1.1 Background Algorithms

At the root of all neural networks is the idea of perceptron Rosenblatt (1958). Developed as a basic supervised classification algorithm, the classical perception consists of a set of input weights based on biological neuron synapses, an activation function that mimics the neuron body activation and single output based on a neuron axon. In the classical implementation, the perception did not exhibit any temporal dependency but was implemented as an instantaneous input to output mapping device. The network composed of a set of perceptrons can classify input signal based on a supervised learning rule and is the basis of most artificial neural networks. By stacking multiple layers of such perceptron nodes, a feed forward network is created that is able to encode more complex functions of the input, but also suffers from more complicated learning process.

A new architecture of Multilayer (DEEP) convolution feed-forward networks has headway in recent years, due to its computational capabilities and efficient learning Schmidhuber (2014). It emerged from the field of image recognition (Zhang et al. (1990)), but has been successfully applied in different fields (Li et al. (2021), Deng (2014)). It represents a subcategory of multilayer feed-forward networks. The two main differences between this and previous neural networks are the reuse of weights and local learning updates. The reuse of weights is achieved by creating a local perceptive window of neuron input, which takes the part of the input space and calculates node output. Sliding this input window to another part of input space allows the calculation of the next output while still using the same node weights. This idea is similar to convolutional image filters used in computer vision. Additionally the output of the one layer does not have to be fully propagated to the top of the network to get weight updates. Instead, the initial local learning rule tries to update weight so that the layer output can be used to most accurately re-represent original input.

Although capable of describing almost any function, the classical feed forward network is not able to express dependency on the temporal aspect of the input stream. The next step in the development on perception based feed-forward networks was the incorporation of its output or previous internal state back into its input. This looping of signal paths as a short-term memory mechanism is a common solution to sequence encoding in neural networks referred to as, recurrent neural networks (RNN) Williams & Peng (1990) and Pearlmutter (1995). Assuming that a classic feed-forward network with enough layers and nodes can approximate any function, a recurrent neural network can technically represent any time dependent neural network architecture. Although possible in theory, this would require both an enormous number of nodes and layers, and, more importantly, learning rules for modifying weights to desired behaviour. In practice, it has been shown that recurrent neural network models have problems capturing long temporal dependencies, because of the difficulty in back propagating error signals in time during learning Hochreiter (1998). These algorithms have been successfully applied in RL, but their scalability is limited, mostly due to the vanishing gradient problem. The erosion of a signal as it is repeatedly fed through recurrent connections limits how long a piece of information can be represented in memory and whether it can be used to influence action selection.

This fundamental learning problem of recurrent neural networks was reduced in specially developed more complex network architectures, such as long short-term memory (LSTM) networks Hochreiter & Schmidhuber (1997) or gated recurrent unit (GRU) networks Cho et al. (2014). These networks are an improved variation of recurrent neural network in which special memory gates are implemented. In such networks, the memory gate state can be acquired from its inputs, stored indefinitely and be read out by the help of additional input trigger lines. This allows long term storage of information, but also requires special learning algorithms. These kind of recurrent neural networks are able to propagate error signals more selectively and thus increase the lifetime of critical information in memory. These type of networks are nowadays state of the art in terms of remembering long term dependencies and working on sequential data.

More recent additions to the family of recurrent neural networks are: reservoir networks in particular Schrauwen et al. (2007), echo state network Jaeger (2001) and liquid state machine Maass et al. (2002). Reservoir networks are essentially a pool of randomly connected recurrent neurons creating a reverberation reservoir. These randomly connected recurrent networks can represent complex temporal input dynamics and the underlying states are typically accessed through a trained feed-forward output layer. The reservoir network is activated with a temporal signal and responds with partially chaotic waves of activation within. The idea is based on the concept of mapping the input vector space, including time, into a multidimensional internal state of network to notice multidimensional interactions between input dimensions. Because the interaction can be read out by simple readout network, and most of such networks' activation is defined by nodes with randomly initializing unchangeable weights inside the reservoir, learning is only required through supervised training to modify weight in the output readout layer. Reservoir networks combine the powerful sequence representation capabilities of recurrent neural networks with the efficient training of feed forward networks. One of the main problems of reservoir networks is that the number of nodes required grows linearly with the required temporal horizon of perception Farkas et al. (2016). The question arises then, how more intelligently initialized weights can improve the performance of this network in terms of memorisation capacity and prediction. Since all weights are randomly initialised the actual utilisation of nodes required for calculating output is minimal. One could imagine some minimal unsupervised adaptation policy that could improve reservoir reaction to the input signal.

Within similar variety of architectures is the Hierarchical Temporal Memory (HTM) network Y. Cui (2015), which uses sparse encoding of signal in the pool of recurrently connected neurons. This network is inspired by biologically plausible activation mechanisms and neuron arrangements. This network architecture consists of sets of recurrently connected neurons arranged in columns. The neurons crossing activation threshold represent a binary and sparsely encoded state of the network. The sparse encoding allows enormous encoding capacity to record multiple states of the network. Although randomly initialised similar to reservoir networks the HTM network follows unsupervised adaptation learning rule that rewards good predictions of future signals. Similar to reservoir networks, the HTM network has a read out layer that can be learned to map an internal state to desired output prediction. Although the network is called hierarchical the elements allowing multilayer arrangement are not yet developed. Regardless, the single layer network allow prediction of both complicated noisy signals like New York weekly taxi demand problem and long temporal dependencies like simple grammar and other long temporal benchmarks. This network represents one of the few examples of unsupervised temporal networks. The network achieves this temporal dependency by the creation of recurrent connections within a pool of neurons. An important part of this network, not implemented in similar reservoir networks, is an input layer called Spatial Pooler, which following unsupervised adaptation, conditions the input stream of data into sparse distribution accepted by the recurrent layer Cui et al. (2017).

Biologically inspired time dependent neuron models make neural network time-sensitive by adding explicit time-dependent features to single neurons within the network (Thorpe et al. (2001), Humble et al. (2012). This approach is explored in spiking neural networks Maass (1997) and even more so, in multiple time scale neural networks Y. Yamashita (2008) and temporal Polychronization networks Izhikevich (2006). This network model follows derivation from biological research of neural cell functionality. The processes involved in natural neural cell activation are simplified into equations and implemented in computer simulation. Compared to other models that are made through utility driven development, these models are trying to isolate crucial elements to neural processes that are solely responsible for information processing. It is not known how much of the multitude of neural activation behaviours are necessary for information processing tasks, but informed search can provide partial answers Sheik et al. (2013). Certain models even allow for hierarchical functional arrangement of nodes Maes et al. (2020), Ghalib & Huyck (2007). Unfortunately, the increased complexity of such models do not allow testing to the degree of other models. This is because increased computational complexity overpowers available resources when huge networks are simulated. Most researchers pursuing bio-inspired networks architectures agree that the human brain represents information not only in the activation of certain neurons, but also in the arrangement of temporal activation. Izhikevich (2006) Wang et al. (2018).

Finally, one of the oldest solutions of introducing temporal dependency into the network is to keep fixed size input sequence segments in a short-term memory in order to extract features across time. This approach has been effective in time-delay neural networks (TDNN) Waibel et al. (1989), FIR networks Wan (1990), Deep Q-Networks Mnih et al. (2015) and temporal convolutional networks (TCNs)

Aaron van den Oord (2016); Bai et al. (2018) You et al. (2019). This approach has a simpler training regime than the recurrent approach, since it can be viewed as a feed-forward network, but it also has much higher memory requirements and a limited time horizon.

1.2 Spatio-temporal encoding

The main feature of the new network presented in this thesis is its ability to deal with spatio-temporal signals. As pointed out earlier, quite often the full extent of information needed cannot be captured in instantaneous input to the network. Instead the input has to observed for an extended time to capture the full scope of the information. Most temporal networks can be divided into three main categories. The first and the simplest approach is to map the input space temporal dimension into additional dimensions of space. This is mainly done by implicit delay lines and memory cells, which augment input space with recordings of previous inputs. Although the idea was early adopted by time-delay neural networks (TDNN) and FIR filter networks. The most limiting factor of this approach is the finite time window which cannot be easily extended without combinatory explosion. This problem was partially solved by Temporal Convolutional networks (WaveNet) by first using a convolutional approach to deal with huge input signal space and, secondly, by using hierarchies and the "time dilatation" of input space. To limit input space at higher layers, the connections in temporal dimension are regularly cut/pruned in the hope that the faster changing dynamics of the signal were already encoded in lower layers. The architecture presented in this thesis uses this model of approaching spatio-temporal encoding by employing short-term memory at the input of each layer. This short-term memory can take the form of classical delay-lines, but the new architecture also introduced novel variable speed analogue delay-lines. These slower analogue delay-lines perform the task of "time dilatation" without a growth in memory requirements. The second, and possibly the most broadly explored, solution to dealing with spatio-temporal information is the mapping of time dimension into the internal spacial representation of the network. This solution is mostly categorized by internal recurrent connections. The architectures exploiting this idea span from simple RNN through LSTM to HTM. The strongest aspect of this approach is the possibility of pre-processing the vast input space into compressed internal representation and storing only the crucial temporal aspects of the signal. Although this might be the most powerful approach for dealing with temporal aspects, the difficulty lies in creating an efficient mapping of input into internal state, which most of the time can only be done by supervised methods, which have to guide network convergence. One alternative to the long supervised convergence of such recurrent networks is the usage of networks with excessive resources to encode spatio-temporal patterns and random initialization, (Reservoir networks and HTM) for a higher chance of a better initial guess. The last approach to dealing with spatio-temporal signals is to equip a network with a temporal reference signal to which a changing input signal can be correlated. This can be done either by creating a recurrent oscillator inside network nodes Maes et al. (2020) or by adding "positional encoding" to the input vector in Transformer Networks Vaswani et al. (2017), Dai et al. (2019). Properties of these networks point to another problem of spatio-temporal encoding, namely temporal alignment. A number of networks consider spatio-temporal signals only with defined start and stop points (TDNN, LSTM) (e.g. simple grammar problem Hochreiter & Schmidhuber (1997)) and even when capable of learning continued problems, they struggle with temporal noise referred to as jitter (HTM) Y. Cui (2015). As an example, Pierris & Dahl (2017) defined the problem as "temporal misalignment" and proposed an algorithm for finding the best alignment in their HSOM architecture. It is understandable that any networks storing some internal representation of past input will be scrambled by abruptly changing input sequence, but one would expect certain resilience to small disturbances in temporal alignment (jitter).

1.3 Self Organizing Maps

Self Organizing Maps SOM are one of the most popular unsupervised learning algorithms Kohonen (1982). This quantisation mechanism creates local topology within a predefined map of nodes based on input space. Each node of the map is characterised by its weights representing the quantisation vector of input space. Nodes of the map compete to represent input space by way of best matching unit function that only selects one node to respond to the current input vector. After finding the best matching node the weights of this node are adjusted to better represent input space. This function creates a "receptive field" within the input space where each node is activated if novel input falls within it. The second important feature of the SOM, which creates topology and increases competitiveness, is neighbourhood function. Each time the best matching node is selected and its weights are updated, the set of neighbouring nodes within the topology of output map are also updated, but with smaller intensity. This decrease of intensity usually follows Gaussian distribution. This update of neighbouring nodes brings their receptive fields closer together, increasing competitiveness, but it also smooths and stretches those fields across input space. Classical neighbourhood functions makes neighbouring nodes on the map represent neighbouring receptive fields in input space. SOM can perform quantization based on map spacial adjacency (neighbourhood function) but also based on a minimum spanning tree Kangas J.A. (1990). Other unsupervised mechanisms, like Natural-Gas architecture can perform quantisation in terms of pure input space Martinetz & Schulten (1991). Although such quantization based on input space might better represent complicated multidimensional signals than the two dimensional topological map, the static two dimensional representation is more accommodating for multi-layer hierarchical representation. There have been a few attempts to make SOM sensitive to temporal information, namely Temporal Kohonen Map (TKM) c. Chappelier & Grumbach (1996) and Recurrent Self-Organizing Map (RSOM) Varstal et al. (1997) and other Mcqueen et al. (2003). These approaches focus on adding temporal behaviour to node activation in the form of leaky integrator dynamics inspired by biological processes. These networks can be seen within the category of networks mapping the temporal aspects of the signal onto the network's internal state. The new architecture presented in this thesis is based on SOM unsupervised quantisation of input space and employs all the mechanism required for it functioning in terms of competitiveness of nodes and local topology. However, the architecture disconnects the sequential function of finding the best matching node and updating weights to insert a variable delay period. This approach encourages the sparseness of node activations across time and enables a resilience to temporal misalignment.

1.4 Hierarchical Reinforcement Learning

The traditional Reinforcement Learning (RL) paradigm operates on discreet time actions and states received from controlling Markov decision process (MDP). Most of the algorithms employed in this classical approach can be extended to semi-Markov decision processes (SMDP) where time between one action and another can be extended by a random variable delay. This expansion allows for the implementation of Hierarchical RL paradigm in which the algorithm does not have to control action down to the "atomic" interactions, but can instead employ learned or pre-programed temporally-extended actions or options Barto & Mahadevan (2002) Kamat & Precup (2020). These extended actions represent short encoded behaviours or skills. This allows for creation for multilayer/hierarchical RL system in which each layer can abstract its decisions to more extended interactions. These hierarchical algorithms provide significant improvement in abstract learning and, presumably more importantly, by allowing more abstract exploration Nachum et al. (2019). Pierris & Dahl (2017) demonstrated how a multi-layer augmented SOM (HSOM) network can represent arbitrary long sequences, and successfully used such networks in an RL context to encode and reproduce robot tactile gestures. That work, however, encoded information in a lossless way, requiring there to be sufficient nodes in the SOMs to represent all the observed sequences. Their approach does not scale to more complex systems or to systems learning from experimentation rather than from demonstration. The new network presented in this thesis shows how it is possible to learn network structures similar to those used by HSOM in a way that has the potential to handle large and noisy data.

1.5 Summary

This chapter has shown ways in which the new network can borrow from the unsupervised algorithm of SOM architecture and the delay-line representation of spatio-temporal encoding of input, in order to acquire long term representations of short sequences of input, which could later can be exploited as options, encoded behaviours or skills in a Hierarchical RL context. All these parts already exist as elements of already established neural networks. The question arises whether a mix of such features would work in combination and produce new anticipated characteristics.

Chapter 2

Methods

In this chapter the main elements of the proposed new neural network architecture will be defined. Parts of the network described here were developed and tested in progression but here they are presented in their final version. First, the mechanisms for capturing the information into the network structure are defined followed by mechanisms allowing its usage in classification and RL contexts.

2.1 Overview

Each layer of the proposed network consists of three main elements: a short-term memory, a long-term memory and a temporal inhibition circuit. Figure 2.1 presents a simplified diagram of the network architecture. Input signal feed to layer is first met by a short-term memory. Its purpose is to capture, store and allow readout of a certain amount of past signal history. It performs the function of mapping time into additional dimensions of space. It allows long term memory to have an overview of the past input without creation of recurrent connections. The short-term memory is implemented as a tape delay-line in the first layers. In higher layers, where signal can be represented as a spike stream, not sampled continues data, and where higher temporal capacity is required, short-term memory is implemented as a simulation of analogue wave memory.

After input augmentation performed by short-term memory, information is passed

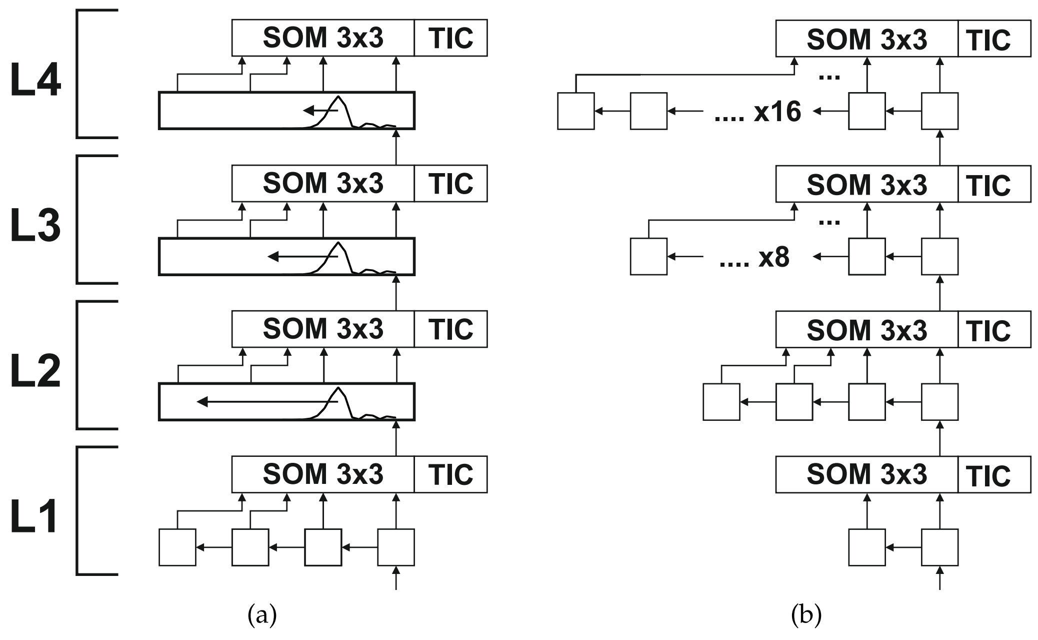


Figure 2.1: Diagram of network architecture. All layers consist of short-term memory, self-organizing map and temporal inhibition circuit. First Layer consist of simple delay line memory but from second layer on-wards short-term memory can be implemented as analogue wave delay line with decreasing propagation speed.

to a modified Self-Organising Map. The SOM is fully connected to all memory units of short-term memory, allowing it to have access to a limited history of past time input. The SOM is responsible for looking at these snapshots of past history and making judgements about recognition and the need for incorporation of this sample into nodes' long-term memory weights. Incorporation of this new memory is consistent with traditional SOM techniques. These include competitive closest match learning and neighbourhood function. The triggering of these winner node learning behaviours is, however, modified and controlled by an additional Temporal Inhibition Circuit.

Temporal Inhibition Circuit (TIC) is closely coupled to the SOM network. It is primarily fed by information of closest match value between SOM input and its node weights. Based on additional local timing information, it triggers positive match and node weights update. Its primary job is to manage the effective usage of long-term memory with respect to the temporal stream of input data. It assists in partial topological temporal signal spanning of input signal. TIC can be seen as an augmentation of traditional SOM quantisation properties which relies on monolithically changing input space. In case of input from short term memory, which is sparse and sharply changing, TIC helps by including temporal dimension into quantisation.

2.2 Short-term memory

A memory is a type of device capable of reading input with the purpose of storing it for latter readout. The presented network architecture consists of two types of memory. Both types work in synergy to each other, in order to reach objectives desired from the network. They differ in terms of triggers required for reading data in and out, and capacity in terms of time and number of stored elements.

The first type of memory enables the network to look at the short term past in a snapshot. It is known as a tape delay line. It is made out of a line of connected memory units. Each memory unit in the line reads data from its previous neighbour and stores it for one time step to be outputted to its neighbour down the line. The first element of the line reads an input signal presented to the delay line. Memory of this input travels down the line until the last unit and then is forgotten. This type of memory was used in first fed-forward networks to create temporal dependency (TDNN). The state of each memory unit can be represented as:

$$x_n(t+1) = \begin{cases} 0 & n=1, I(t)=0 \\ 1 & n=1, I(t)=1 \\ x_{n-1}(t) & n>1 \end{cases} \tag{2.1}$$

Given $I(t)$ is the input to the first memory unit, $x_n(t)$ is the state of the n-th memory unit, the input value gradually travels down the memory line while the output $O(t)$ represents all memory unit at once

$$O(t) = [x_1(t), ..., x_n(t)] \tag{2.2}$$

When looking at all units, memory of input from the last few past time steps can be uncovered, at once. This type of memory is a way of mapping dimension of time into additional space dimensions. This type of mapping is only a means to allowing the functionality of the second long-term memory.

Tape delay line is a method of including past input into current representation. A problem arises when long temporal capacity of this memory is required. Since each block can only store data for a one time step, increasing temporal capacity can only happen by increasing the number of the memory blocks. This creates two problems. The first is, doubling the memory capacity doubles the computational resources required to operate it. Secondly, it also increases the number of readout lines.

There are numerous ways to solve problems of extending tape delay memory time capacity. They include modifying memory unit to hold memory for more than one time step, skipping time steps at which new input to memory is read and only reading out partial output from the too long memory line. However all these methods suffer from the problems associated with discrete time steps. In discrete time space, events can only happen at the certain time point so using any of described methods might warp the time by down-sampling of the original signal. When performing such an operation, re-sampled signal loses high frequency components. This would mean losing exact input signal timing information or the whole signal completely. Additionally, down-sampling high frequency signal can introduce noise associated with not fully rejected high frequency components.

At higher layers of the network, the signal is represented as stream of SOM node activations that can bee seen as spikes of activations. After converting the signal to spikes by first layer it is easier to think about constructing memory with higher temporal capacity. The spike signal is bounded in value. It can be also be bounded in number of spikes per given time. This opens possibilities to creating memory specialized in storing only the signal conditioned in this way. One way of increasing the temporal capacity of the memory like delay line is to slow down the speed of data propagation. This, of course, is not possible in the basic implementation of delay line memory units and requires more complex single memory unit behaviour.

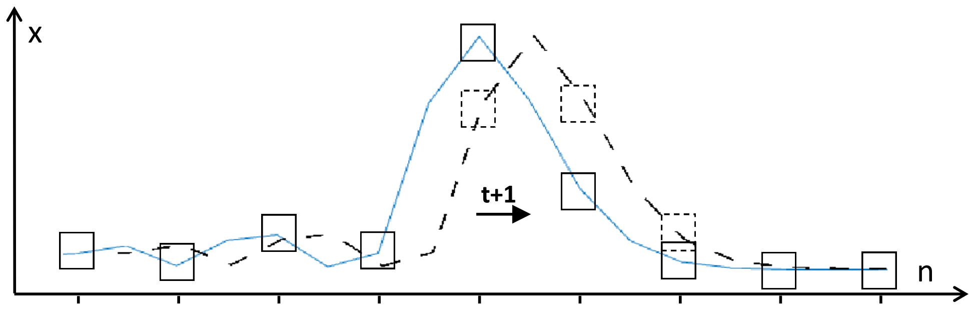


Figure 2.2: Wave propagating through analogue delay line.

Most real world physical signals are propagating with limited speed dependent on the medium. This phenomenon was used in the creation of the first refreshable computer memory type using mercury delay line. This type of memory was a partial inspiration to the implementation presented here of a new type of short-term memory. Simulation of simplified dynamics of a physical medium can recreate signal propagation with limited speed. This can be done by simulating a string of points of mass connected by ideal springs simulated as a second order systems. Disturbance at one end gets propagated through the line to the other end. The idea of this model is presented in Figure 2.2. Parameters of the simulation influencing speed of propagation can be changed to suit the requirements of temporal memory capacity. The temporal capacity of such memory cannot be increased without a cost. Without an increase of a number of simulated particles temporal resolution, and an amount of independent spikes travelling and being memorised at once, will be limited. This type of memory allows the network to increase memory temporal capacity at a cost of reducing its temporal resolution. Since spikes propagating through simulated physical memory like that are carrying energy, which gets transmitted through the line, they have non-zero width (see Figure 2.2). This property allows modelling of temporal noise (jitter). Such a wave has rising and falling slope artefacts which, when remembered, can model the inconsistencies of spikes occurrence.

Implementation of such wave memory follows simple physical dynamics equations. Each memory unit is simulated as point of mass and is allowed uni-dimensional movement perpendicular to wave propagation. Units are connected by ideal springs. To stop back propagation and past resonance, movement of the mass is stopped behind the wave when it reaches zero point. This requirement might resemble neuron action potential propagation, but is implemented here from a purely utility driven perspective. Finally, the last unit has additional damping parameter to stop wave reflection. Full equations of motion can be found below:

$$dx_n(\tau) = [x_{n+1}(\tau) - x_n(\tau)] \tag{2.3}$$

$$v_n(\tau+1) = \begin{cases} v_n(\tau) + Cdt\,[dx_n(\tau)] & n = 1 \\ v_n(\tau) + Cdt\,[dx_n(\tau) - dx_{n-1}(\tau)] & L > n > 1 \\ v_n(\tau) - Cdt\,[dx_{n-1}(\tau)] - Bv_n(\tau) & n = L \end{cases} \tag{2.4}$$

$$x_n(\tau+1) = \begin{cases} 1 & n = 1, I(t) = 1 \\ 0 & x_n(\tau) < 0 \\ x_n(\tau) + Cdtv_n(\tau) & \end{cases} \tag{2.5}$$

Where $x_n(\tau)$ is a memory node state $dx_n(\tau)$ is neighbouring nodes position difference and $v_n(\tau)$ is nodes' rate of change. As previously, $I(\tau)$ is the input to the memory. The equations include set of parameters like last unit damping ratio $B = 0.14$, simulation interval $dt = 0.1$ and propagation speed $C = 1$ which can be changed depending on memory temporal capacity requirements. To improve smoothness of dynamics, the equations' iterations (τ) are run 10 times faster compared to networks' iterations (t).

$$\tau = 10t \tag{2.6}$$

Additionally during readout, to narrow the propagation wave, the nodes' positions are sub-sampled, such that output vector $O(t)$ reads the position of every other node. As a result the output vector is defined as per equation:

$$[o_1(t), ..., o_m(t)] = [x_1(10t), ..., x_{2m-1}(10t)] \tag{2.7}$$

The code implementation performing the short-term wave delay line memory function can be found in Appendix A.

2.3 Long-term memory

Although the short-term memory is capable of storing the input data, it does not discriminate between important information and noise. That is the purpose of the second, long-term memory. There are two commonly recognized ways of discriminating and categorizing information. The first one is based on additional supplementary signal and is referred to as supervised learning. The second is based on probability distribution clusters persistent in the input space and is referred to as unsupervised learning. In the set of problems presented to autonomous systems trying to discover emergent representation of perception, the additional supplementary signal allowing supervised learning is absent. Therefore local unsupervised clustering based on probability distributions is the only solution.

The way to solving the unsupervised learning challenges of clustering based on probability distribution involves utilising vector quantisation methods. It is a way of finding characteristic vectors describing clusters of probability distributions of input vector space. In this method, the fixed number of vectors, sometimes named nodes or code-book, competitively fight for the representation of input space. At each input sample the closest node is selected and updated in order to better represent input space clusters and decrease overall mapping mismatch. By keeping the number of vectors fixed, or slowly growing, the vectors can only represent general probability centres that correspond to quantised input space. By creating a map with local interaction between quantisation nodes Kohonen (1982) proposed a way of creating an emergent topological arrangement in which the nodes closest on the abstractly created map of nodes are also close in the input space. When there is no interaction between the nodes themselves, except for competitiveness in representing input space, the mapping can be referred as topological zero-order.

There are four crucial elements required in creating topological unsupervised learning, as defined by Kohonen's Self-Organizing Map. The first element is an array (map) of processing units, nodes, representing quantisation vectors through their weights. The Array or Map is an arbitrary arrangement of nodes in activation output space. The second element is a discrimination function that compares the node weights to input vector and is a way of finding the best matching unit (BMU) within the array of nodes based on this discrimination function. The third is a network of local interactions between nodes in a small neighbourhood of the array and the fourth is the function of adapting nodes in a neighbourhood to better respond to presented input. All those elements have to be present in order to create topological unsupervised learning, but the exact implementation can vary depending on design.

All of the experiments were performed using a square map of nodes. This type is map is the most common and creates the simplest distribution of nodes with respect to neighbourhood function, since all nodes have evenly defined positions. Since the input space for the problems that network was presented with, was mostly highly multidimensional, the square network appeared to give the most flexibility to adapt.

There are two most common discrimination functions used to assess similarity between input vector and node weights. Namely Euclidean distance and dot product. The dot product function calculates the sum of products of all components between two vectors, which equates to the product of two vector lengths and the cosine of the angle between them. The more similar the vectors, the bigger the dot product between them. The dot product function is dependent on vector length, therefore it has no maximum value, when used as discrimination function between input vector and SOM weights. This makes it more difficult to track the progress of network convergence since weight vectors change length. Because of this, the measure of Euclidean distance between the input vector and node weights was used as discrimination function in this network. The Euclidean distance between two vectors is simply the length of the vector that is the two vectors' difference. Although the more the vectors are dissimilar, the bigger the distance can be, when vectors become identical the distance drops to zero, independent of the number of vector dimensions. This makes is easier it track network performance during training and compare different networks with different input vector sizes. The equation defining the discrimination function can be found below:

$$BMU(t) = \underset{i}{argmin}(||O(t) - W_i(t)||_E) \tag{2.8}$$

Where $W_i(t)$ are the weight of a particular SOM node, $O(t)$ is a input vector and $BMU(t)$ is a best matching node.

Finally, a neighbourhood function with the influence in the shape of Gaussian function was applied in the algorithm described. The main reason for neighbourhood function is to bring vectors together and add inactive nodes to the pool of existing probability densities. By bringing vectors together the function stretches and smoothes out the map over the input space, creating a topological mapping of nodes, where nodes that are close on the network map are also close in the input space. Also without the neighbourhood function, a naive node with undeveloped weights at the beginning of learning could never be chosen as a best matching node and never have its weight updated toward the possible probability cluster. With the neighbourhood functions applied, whenever a single node is shifted towards a probability cluster, all neighbouring nodes are shifted too, making them more likely to be activated in the future. Despite this, in the class of problems that this research presented the SOM network with, very sparse representation and input space with very dense probability distribution, forces some nodes to be outside of representation, effectively making them unused. In this situation, the question emerges whether the enforced topology is necessary or useful, but since the neighbourhood approaches zero order at the end of the session, the neighbourhood function in theory should not be handicapping the learning. Below is the equation that describes the neighbourhood function:

$$h_{ci}(t, BMU) = \exp\left(-\frac{(x_i - x_{BMU})^2 + (y_i - y_{BMU})^2}{2\sigma(t)^2}\right) \tag{2.9}$$

Where $h_{ci}(t, BMU)$ is a neighbourhood function, x_i, y_i is a position of a node on the SOM map, x_{BMU}, y_{BMU} is a position of the best matching node and $\sigma(t)$ is a neighbourhood reach.

To make SOM converge to the desired representation, the learning rate for weight updates is started with considerable value and is gradually reduced, slowly freezing the weight in the final representation. The decaying learning rate allows for a change from courser to finer changes to weights, allowing for fast initial convergence to general representation and later fine tuning to better matching. Similarly the neighbourhood reach is shrunk during training. Decaying reach of neighbourhood allows for wide reach and the bringing of all nodes' weight vectors into the general area of probability distribution at the start of the training, while simultaneously reaching almost zero-order topology with narrow reach at the end of learning. The following equation describes the progression of decay for learning rate and neighbourhood reach:

$$\begin{aligned} \delta(t) &= \exp\left(-D\frac{t}{T}\right) \\ \alpha(t) &= A\delta(t) \\ \sigma(t) &= S\delta(t) \end{aligned} \tag{2.10}$$

Where $\delta(t)$ is a decay value, D is a decay rate, t is a time step, T is a number of all time steps. $\alpha(t)$ is a learning value, A is a learning scaling factor, $\sigma(t)$ is a neighbourhood reach value and S is a neighbourhood reach scaling factor.

Putting all of the elements together, the final weighs update for a particular SOM node is as follows:

$$\begin{aligned} &W_i(t+1) = \\ &W_i(t) + g(t)\alpha(t)h_{ci}(t, BMU)[O(t) - W_i(t)] \end{aligned} \tag{2.11}$$

Where $W_i(t)$ are the weights of a particular SOM node, $O(t)$ is a input vector, $\alpha(t)$ is a learning rate, $h_{ci}(t, BMU)$ is a neighbourhood function, and finally $g(t)$ is a gating signal from a temporal inhibition circuit that inhibits learning.

Random initialisation of weights helps with randomising the initial unfolding of a two-dimensional map into multi-dimensional input space. As far as the mapping is never unique with multiple possibilities of unfolding, a random initialisation allows to try different initial approaches. Thus, the weights of the SOM are initialised with very small random numbers. Weights much smaller than expected to develop do not overpower and disturb the development into their final configuration.

The topological mapping can be argued to provide advantageous behaviour in the context of augmented autonomous systems perception signal. The topological mapping proposed by Kohonen has some biological basis and could be advantageous when used to help with action selection when the map is employed in a RL context. Although as later shown in the experiments, the topological mapping of nodes does develop in this modified SOM implementation, the closeness of nodes with similar weights on the map is not utilised in any way. Thus, the creation of topology is not crucial or required for the purpose of the SOM used in this implementation. The creation of topological mapping is only a by-product of the neighbourhood function. The neighbourhood functions' purpose is not only to create topology, but also to recruit and move unused nodes into the probability cluster, where they have a higher probability of being activated. Without this function one random node may take over all representation of the input distribution and stop the competition of nodes in the fight for input representation. Simultaneously, a strong neighbourhood function may not allow nodes to individuate on particular clusters of representation. It is therefore crucial that the effects of this function decay fast during the learning, so that the nodes can be recruited and moved into the vicinity of general input data cluster, but then let go to individuate without being effected by their neighbours. In the current implementation, the neighbourhood function treats all node weight components independently and applies neighbourhood effects across the same weight component for all nodes in the map. Yet since the input vector is a composition of time augmented signals read from short-term memory, it can be seen that nodes weights are not made of independent components. Thus, some more elaborate neighbourhood function that could take this time dependency of node weight components into account creates interesting an avenue for development, but was not explored as part of this work. Such a neighbourhood function could be seen as creating the topology not only in the spatial dimension but also in the temporal dimension.

SOM is a type of network which by the use of unsupervised learning can create a discretised reproduction of input space. The implementation of SOM here is fully connected to short-term memory at each layer of the network. When fully connected to the short-term memory type it might be expected to learn and represent chronological patterns of the input signal. There are a lot of unknowns when forming a memory of temporal signal; like when it should start, when it should end, what it should contain, how to store it, and how to know whether similar memories already exist. At each time step new input is added to the delay line, previously stored memories shift back and the last is forgotten. Although in the memory representation of the past in vector space has changed drastically, the amount of information about the signal pattern has not. Most of the new short–term memory view consist of the same portion of signal history. Unfortunately, SOM mechanisms require more monolithically changing input space. Thus it is inefficient for the SOM, which is fully connected to the delay-line memory, to try to discriminate input representation of delay line at every time step. This creates a need for additional discrimination based on timing to decide when the representation of past stored in delay line has changed enough that it is required for SOM to incorporate it into its vector quantisation. This function is mitigated by additional temporal inhibition circuit. Standard equation of SOM is only augmented by an additional gating signal which controls when the learning should take place, the rest of mechanisms are left intact.

The full code implementation performing the Self-Organising Map function can be found in Appendix B.

2.4 Temporal Inhibition Circuit

In the problem that is being addressed in this study, the temporal sequence is being encoded by a Self-Organizing Map. In original implementation, Self-Organizing Map learns and quantifies the input vector at every time step. Having a fixed number of nodes, the number of unique time steps SOM can encode will be the same as the quantity of nodes. We can regard this value as temporal or sequence length capacity. Since in the implementation presented the input vector to SOM is augmented but short term memory, in the form of delay lines, the temporal capacity of each SOM node gets extended by this short term memory length value. The resulting temporal capacity of such a network can be approximated by the product of short term memory length and SOM nodes quantity. Unfortunately, in classical form of the SOM learning paradigm, SOM nodes have to be activated at every time step. Thus information (sequence snippets) contained in the short term memory will overlap between nodes and the memory capacity of such a network still only depends on the quantity of nodes. Consider the example presented in Figure 2.3A where the short repeating sequence of length 7 is being encoded by nodes augmented by a short term memory of 4 time steps. When nodes are activated every time step the number of nodes required to encode this sequence is the same as sequence length, while the nodes memory of the sequence has considerable overlap between.

By knowing the length of short term memory augmentation, the SOM matching

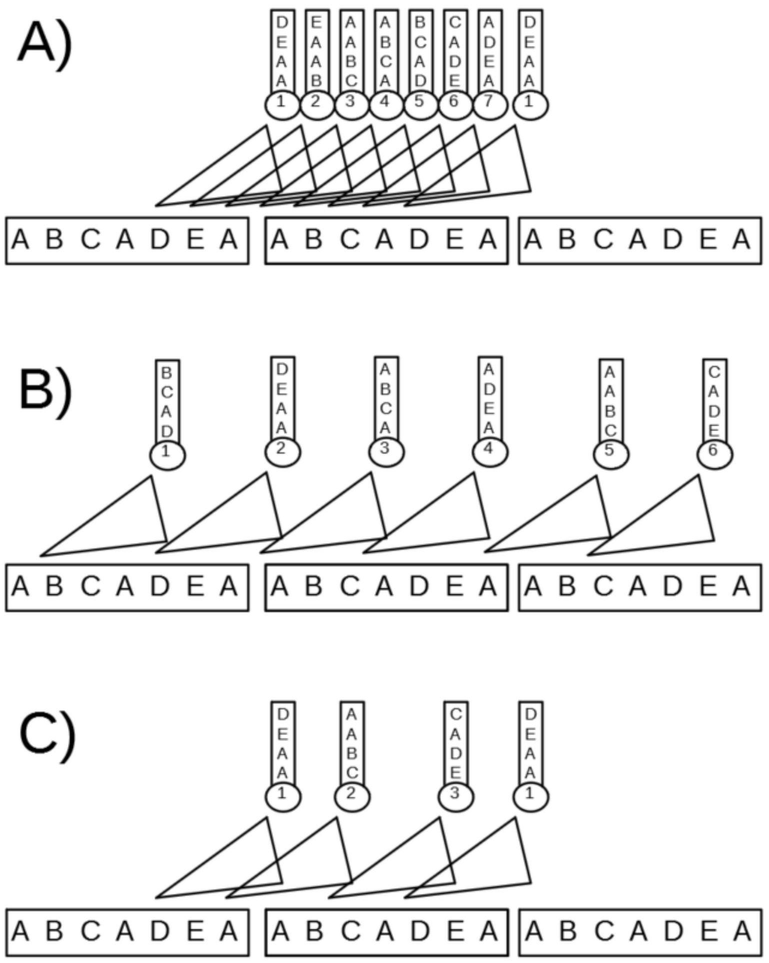


Figure 2.3: Example encoding of repeating sequence of period of 7 steps ("AB-CADEA") encoded by SOM augmented by 4 time steps delay line. In normal activation paradigm (A) the nodes of the SOM (1 through 7) are activated every time step and sequence chunks encoded in the weights overlap. When SOM activation is time dilated (B) by three steps the sequence chunks stop overlapping but lack of synchronisation means that SOM still needs 7 nodes to encode whole sequence. Only when activation is unevenly dilated in time can the whole sequence be encoded by only 3 nodes (C).

and learning functions can be regularly halted for a few steps to allow short term memory to be filled with more new data samples. This could extend the overall network sequence length capacity, as long as the halting period is not longer then the short term memory length. This approach can be regarded as down-sampling of the input signal and SOM activation. A similar technique is used on higher layers of TCN networks and is referred as, "time dilatation". This solution, although simple, suffers from the problem of synchronisation with the input sequence. When sequence is not synchronised and its length is not divisible by a halting period the activation of the nodes changes every time the sequence is presented (see Figure 2.3B). In this example, a halting period of three steps makes the sequence of length 7 appear different each time it is presented. This results in activation of different SOM nodes every 21 steps when the halting and the input sequence re-synchronise again. Although this halting technically extends the network sequence length capacity, in the example presented the network still requires 7 nodes to represent the sequence and, additionally, the representation of nodes activation is artificially extended to a 21 step sequence.

The only solution to the presented problem of activation is to introduce a type of soft halting function. This function would have to introduce halting to extend SOM sequence length capacity, and also allow for the re-synchronisation of nodes' activation every time the sequence repeats by also allowing shorter halting delays. Consider the activation on Figure 2.3C. Although the one node exhibits a 3 step halting period, the other two nodes' halting period is shortened to 2 steps, making the overall SOM activation repeat and synchronise with 7 step sequence. In this example, the sequence of 7 is encoded by activation of only 3 SOM nodes every 2 or 3 time steps, creating the desired temporal and spatial compression of the sequence.

To produce a mixed activation pattern of SOM nodes, the Temporal Inhibition Circuit (TIC) was created. It modifies default patterns of matching and learning to accommodate best usage of memory resources. In its classical implementation, the SOM algorithm performs a function of finding a best matching node and adjusting its weights and those of its neighbours at every signal sample. The inhibition circuit operates by breaking this link; still performing the function of finding best matching node, but the decision to adjust weights in this particular time step is triggered by TIC only at appropriate moments. This triggering of learning, referred to as node activation, is conditioned on two aspects. Firstly, the time since the last node has been triggered and secondly the strength of the match (Euclidean distance) between input and current best matching node weights. The "time since last node activation" part acts as, mentioned earlier, “time dilatation” or, “refractory period”, forcing nodes to spread their activation in time. This part does not strictly forbid activation of any node for certain time. Instead it enforces softer, “refractory threshold” which puts a barrier on best matching node Euclidean distance value "goodness" before the node can be activated.

The exact algorithm for the temporal inhibition circuit can be implemented in a few steps. Firstly depending on the desired performance two static variables are defined: the maximal refractory period (P) and minimal refractory level (R)(see Figure2.4). The TIC functioning is based on two main global variables: a map of values representing time since last activation corresponding to each node of the SOM map and a single global refractory threshold that defines current threshold below which new matches are accepted. At each time step the new input vector is queued and delay lines are updated. The best matching node is found within all SOM nodes. If best matching node Euclidean distance is smaller then refractory threshold, and this node has not been activated in last P steps, the node enters activation state. This state represents a positive match so the nodes’ weights are updated as per standard SOM update strategy with neighbourhood update. In this activation state, nodes’ variable for time since last activation get reset. Simultaneously, the global refractory period gets set to a more rigorous (lower) value of R. Finally, in this activation state, this node activation gets advertised as one-hot map to the next layer above. Independently whether any nodes have been activated, each time step the values for all nodes’ time since last activation get increased. Additionally, when time since last activation for all nodes gets bigger then the P value the global refractory threshold gets relaxed to infinity to allow any node to get activated. Figure 2.4 describes the expected behaviour of the TIC in visual form. This implementation helps to span SOM weights most efficiently across input sequence, but is also flexible enough to allow re-triggering of already memorised patterns. Only positive match and weights adjustment point to signal pattern detection event and is fed as a positive node activation to the next layer. The behaviour of the temporal inhibition circuit can be defined by a set of equations. First is a definition of map representing the time since each node

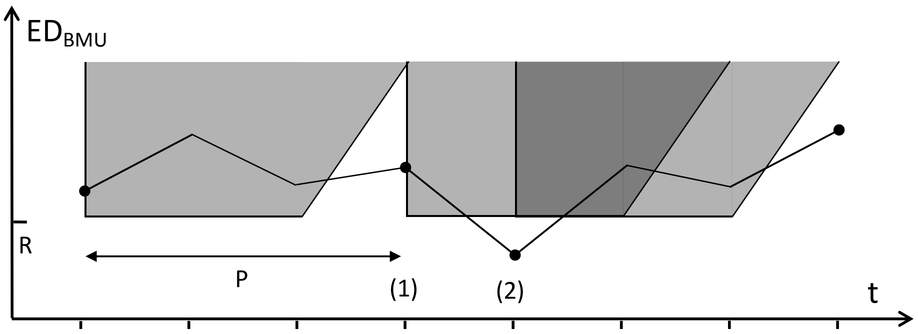


Figure 2.4: Behavior of temporal inhibition circuit. After each node positive activation (full dot) TIC creates period of inhibition P at which any BMU node which has ED bigger then some defined value R is inhibited. Any node with better match can override inhibition at any point and trigger activation and learning (2) else the network has to wait till inhibition elapses (1).

was activated:

$$A_i(t+1) = \begin{cases} 0 & g(t) = 1, i = BMU(t) \\ A_i(t) + 1 \end{cases} \tag{2.12}$$

In the next step global refractory period value and its longitude is defined based on the activation map:

$$r(t) = \begin{cases} R & \exists_i, A_i(t) < P \\ \infty \end{cases} \tag{2.13}$$

Next, based on activation map and refractory value gating output is calculated:

$$g(t) = \begin{cases} 1 & ED_{BMU}(t) < r(t), A_{BMU}(t) > P \\ 0 \end{cases} \tag{2.14}$$

Additionally, gating output and BMU value are combined to create one hot representation of the activation map as an input to the next layer:

$$G_i(t) = \begin{cases} 1 & g(t) = 1, i = BMU(t) \\ 0 \end{cases} \tag{2.15}$$

At the extreme values of the refractory period (P) and the refractory threshold (R) all of the activation patterns presented in Figure 2.3 can be recreated. When the period is set to zero and the threshold to infinity the pattern of activation presented in Figure 2.3A will ensue. When the period is set to 3 and the threshold to zero, the pattern of activation in the Figure 2.3B will be visible. By setting the values in between those extremes the pattern from the Figure 2.3C can be recreated.

The exact implementation of the code performing single layer network functions, joining short-term memory, self-organising map and temporal inhibition circuit can be found in the Appendix C.

2.5 Multi-layer connections

One layer can find statistical redundancies in the signal only to the extent of its short-term memory. This memory can be enlarged, but when the signal is lacking statistical redundancies in repeating patterns, limiting the size of the memory nodes will prevent parts of the node weights to converge on single representation. Even in such cases, partially converged layer memory can still detect smaller signal redundancies and react to them by positive match activation. This activation can be converted into an activation signal that can serve as a input signal to the next layer. This layer activation signal will be seen as a quantised and down sampled version of the original signal. Provided the signal exhibits some temporal redundancies, new representation will have a lower density of information and will be compressed. This allows the next layer to have a longer short-term temporal window without overwhelming the quantisation mechanism. These properties allow the stacking of such networks with increasingly longer temporal views. The multilayer connection can be simplified to the following:

$$L1\begin{cases} O^{L1}(t) &= DL(I(t)) \\ G^{L1}(t) &= MSOM(O^{L1}) \end{cases} \tag{2.16}$$

$$L2\begin{cases} O^{L2}(t) &= WDL(G^{L1}(t)) \\ G^{L2}(t) &= MSOM(O^{L2}(t)) \end{cases} \tag{2.17}$$

Where $DL(I)$ is simple delay line implementation, $WDL(I)$ is a wave delay line, and $MSOM(O)$ is a Modified (TIC augmented) Self-Organizing Map.

Depending on the amount of the frequency components of input signal, each layer can have exponentially increasing short-term memory capacity without an increase of long-term memory nodes numbers. Together with the property of limited analogue delay line memory requirements the addition of new layer and, therefore, doubling of temporal signal spectrum can inflict only linear growth in computational expenses.

The exact implementation of how multiple layers of such network can be joined to encode complex repeating signals can be found in Appendix D.

2.6 Classification

The SOM architecture is designed to create quantisation of input space in the form of a topological map based on frequency of occurrence and spatial proximity. It produces the unsupervised compression of input space in form of quantization vectors (QV). It has also been discovered that the input vector can be augmented by additional elements to create a type of associative memory Kohonen (1989). After the formation of a SOM map trained on input signal composed of two elements, the search through the QV can produce a simplified/compressed association codebook. The size of the network can define the coarseness of the associations. Since the matching and neighbourhood functions act on the whole input vector, the relative size (Euclidean length) of the associated parts can shift the importance of clustering from one part to another. In particular, the use of this type of associative encoding for classification tasks was explored using augmentation of input vector by class identity unit vector Kangas J.A. (1990).

$$I(c) = [i_0 = 0, ..., i_c = 1, ..., i_n = 0] \tag{2.18}$$

$$O_{train} = [S_{train}, \gamma I(c)] \tag{2.19}$$

Where $I(c)$ is a class identity vector of class c of n number of classes. O_{train} is an input vector during training composed of data sample S and identity vector I scaled by γ parameter. This one-hot vector representation of the class identity added to input vector gets associated in the node weights through SOM unsupervised quantisation.

Later search for the best matching unit with the test input sample can yield a class identity. Since the test input vector has unknown class identity at the moment of the search, Euclidean distance calculations during matching have to ignore dimensions of the identity component and empty identity vector has to be added to the input vector composition:

$$BMU_{test} = \underset{i}{argmin}(\|[S_{test}, [0,...,0_n]] - W_i\|_E) \tag{2.20}$$

$$C = \underset{j}{argmax}(I_{BMU}(j)) \tag{2.21}$$

Where BMU_{test} is a node closest matching the test sample S_{test} (ignoring identity part) and C is a predicted class of test sample. This algorithm might be viewed as an implementation of k-nearest neighbours on simplified and compressed data sample. This compression of the dataset speeds up classification and allows for the generalisation of clusters. Because of properties of the SOM neighbourhood, some nodes can be placed on the border of the class divisions. These nodes can then appear close to some borderline test samples and its identity vector part will be blurred, showing partial affiliation with multiple classes. To guard against this problem, additional checks for clear class identity was shown to improve classification results (Kangas J.A. (1990)). One of the down sides of using unsupervised quantisation algorithm for classification task, is that quantisation has no inherent goal of placing QV in places where they could help the most in the classification task. The QV are distributed evenly, based on statistical frequency of occurrence and input vector proximity which might not be desirable for dataset with uneven frequencies and spatially distant classes. See the example visualization of not effectively placed nodes in Figure 2.5.

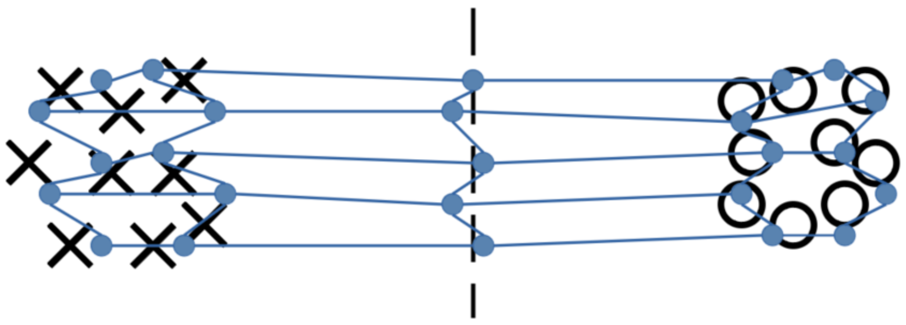


Figure 2.5: Visualization of 5x5 classification SOM network trained on two clusters of data (x and o) with additional class identity vector. The left 10 nodes captured the first data cluster (x), while the right 10 nodes captured the right (o) cluster. The center 5 nodes where pushed by neighbourhood function to the border of the class division and do not help with classification. They could be recognized by blurred class identity vector with confidence lower then 75%

2.7 Prediction, Value estimation and Policy for Reinforcement Learning

All problems of control can be abstracted to the general formula of controlling some "black box" system. A black box system follows certain unknown internal dynamics: it can accept external input and allow observation of some internal states. The goal of a controller is to produce the input in the form of actions to drive the system to the desired state. In special cases when the desired state is hard to define, the desired outcome or path can be more easily described by a reward signal. In these cases, the controller designer's goal is to find optimal actions conditioned on an observable internal state to maximise received rewards. Finding such a controller/agent and its optimal actions is a basic challenge in field of Reinforcement Learning (RL).

$$p(s_{t+1}|s_t, a_t) \tag{2.22}$$

$$r(s_t, a_t, s_{t+1}) \tag{2.23}$$

$$\pi^*(a_{t+1}|s_t) \tag{2.24}$$

In a basic definition of the RL problem the system follows known dynamics described by transitions function p. The transition functions defines probability of system reaching state s_{t+1} conditioned on previous state s_t and action taken a_t. The system produces immediate rewards r, which are calculated based on internal state or state transitions. The goal of RL algorithm is then to find optimal policy π^* function, which produces the probability of taking actions a_{t+1}, conditioned on current state, such as to maximise received future rewards when following such policy. Such systems with defined state transitions and observable states are referred to as Markov Decision Processes (MDP). A solution to the RL problem is to create Value estimation for each of the internal states. This Value function represent discounted expected reward for being in that state and following optimal policy. Knowing the value function and state transitions the optimal policy would choose actions following/traversing states with highest expected reward.

Given the ideal model of the system, in the form of a transition function, algorithms defined as Dynamic Programming (DP) were shown to be able to iteratively converge on optimal value estimations Bellman (1953). For each state they averaged immediate and expected rewards for transitioning to the following state, weighted by probability of each transition given the policy. By iteratively looping through all states, value estimates converge to true values. In a process referred to as Generalised Policy Iteration (GPI), first the state values are evaluated for a given suboptimal policy. Subsequently new, better policy is derived from these value estimates. By iterating between these two operations, the value estimates and policy has been proven to converge to the optimal solution.

Methods referred to as Monte Carlo (MC) for their random sampling, can evaluate state values only from samples of experience. In cases when there is no knowledge about systems dynamics, it is still possible to get traces of actual system behaviour, either as a recording of real system or just its simulation. These traces represent samples of experience of interacting with such systems in the form of state, action and reward chains. For such traces to be useful in predicting state values, they need to lead to some terminal reward. Thus, such traces can be derived only from episodic problems. MC methods evaluate the value of each state in a trace and then average the values of each state across traces. Since the system transition function is unknown, the state value estimation does not allow a new policy to be derived. Thus, more often MC methods are instead employed to derive state-action value estimates, which represent expected future reward for being in certain state and taking certain action. State-action values can be directly utilised to obtain greedy policy by finding action leading to best expected reward for each state.

By combining the ideas of Dynamic Programming and Monte Carlo methods, the algorithm of Temporal Difference (TD) was formulated. By allowing us to estimate value of current state based on estimates of other states (DP contribution) and to define state transitions based on random sampling from experience (MC contribution), the algorithm achieved the most universal scope. It can be applied to continues problems without known dynamics and without recording of experience traces. Although efficient, TD estimation of state value without known transition function does not allow for extracting policy. Thus, a more detailed variation of TD algorithm was devised, known as Q-learning, which estimates values of state-action pairs known as Q values. The general formulation of Q-learning algorithm can be define by its Q table TD update.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a(Q(S_{t+1}, a)) - Q(s_t, a_t)] \tag{2.25}$$

At every time step the entry to the Q table entry indexed by state and action $Q(s_t, a_t)$ asymptomatically aims with speed defined by α to reach the target value by taking small update steps. The update change known as temporal-difference is a difference between the current estimate of the state-action value, intimidate reward r_{t+1} and discounted by γ best possible state-action value reached in the next step Watkins (1989).

One difficulty with the algorithms presented is the assumption of full observability of the system. For complicated systems, internal state is very rarely fully observable, as the controller/agent only has access to subset of internal states called observations. These problems are referred to as Partially Observable Markov Decision Processes (POMDP). Essentially, any single observation is ambiguous in describing internal system state, but when multiple consecutive observations are connected the internal system can be discovered. Additionally, the state space of advance problems becomes untraceable, because a number of discretised dimensions of state is impossible to approach in a table like fashion. An additional obstacle in the presented abstraction is that no system can be fully isolated from the outside world, and therefore any system will have uncontrollable outside disturbance, which will affect internal state. These disturbances can usually be modelled as noise. In simulation problems, these disturbances are also sometimes an effect of random exploration.

In the final part of this research, we argue that some of those problems can be mitigated when the presented network is added as an unsupervised state decoder. When trained on sequences of observation, action, reward triples from the RL problem, the network can discover repeating traces of problem dynamics. When discovered traces are encoded in the long term memory they can be accessed as both, traces of interaction as in the Monte Carlo method and as transition function. Additionally, depending on the problem, when remembered traces of observation are longer than the POMDP order, the memory traces can start to be regarded as actual discovered states of the RL problem, not only observation. See Figure 2.6 representing conceptual diagram of such utilisation of the new architecture.

When an initial solution to the RL problem is unknown, then random policy can be employed to explore the problem. The network can then observe the output of such random policy. The weight developed by encoding such interactions can be regarded as encoded historical traces of interaction with the RL problem. From this point, to improve on random policy the historical traces have to be differentiated based on their expected reward. Two ways have been developed to assign such value to the network nodes (encoded traces). The first method, referred to as Feed-Up, searches rewards in the traces of interaction encoded in network

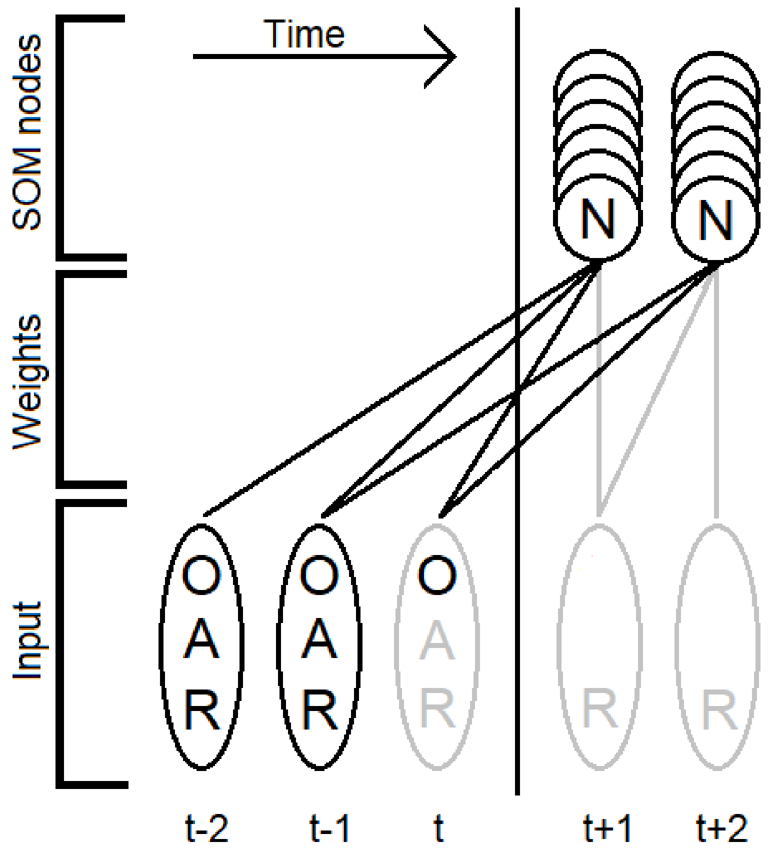


Figure 2.6: Diagram of presented architecture utilised in RL context. The network is trained on observation, action and reward sequence. By using SOM as temporal associative memory the node weights can be used as approximation of transition function and searched for actions maximising expected future reward.

node weights and reassigns them to the nodes representing those traces on the higher level. This method resembles the Monte Carlo method, in that it searches traces for rewards and tries to update all proceeding states with discounted value. The second method, referred to as Temporal-Learning, redefines the temporal-difference equation in that instead of updating the entries to Q-table, the values assigned to network nodes are updated based on their activation pattern.

To utilise the discovered node values in better action policy, expectation of which network nodes may be activated have to be calculated. This can be done based on the algorithm for classification presented earlier, which employs SOM network as a associative network. By leaving some parts of the weights blank, the network matching function can calculate partial association to past context stored in delay lines. The network has to consider node activation in a few future time steps and consider context information with different time shifts. See diagram of building of such association of input for the future time activation predictions in Figure 2.6.

When both possible future development of activation and node values are known, the decision of which possible node trace to follow can be made. Although more sophisticated combination of those two values could be developed, the simplest solution was to chose the node with highest product of probability and expected value. After choosing the node that represents the most favourable future, the policy is to just choose the action stored in its trace weights.

2.8 Implementation

All of the elements of the new neural architecture described above were developed gradually and implemented in MATLAB scripting language inside a dedicated MATLAB environment. Although this scripting language does not provide the fastest execution time, in this development, the flexibility of quickly implementable changes was more important. Additionally, the MATLAB environment allowed easy visualisation of the important parameters during and after network training. Each part of the network was developed as a separate class object in order that it could be more easily incorporated and ported into each experiment, and to providing sufficient level of abstraction. The implementation code for each of the main network elements can be found in Appendix A.

2.9 Parameterisation

As the goal of the development was not to find the best and fastest final performance of the network, but rather to prove that the network could achieve the desired goals of encoding, the task of finding the best hyper-parameters was not of crucial importance. Nonetheless, the hyper-parameters of the network had to fall within certain values for the network to converge and perform its functions. The new network's high number of hyper-parameters defining its performance can be divided into two general classes: the parameters defining network's encoding capacity and the parameters defining network's learning and convergence. Since the numbers of layers, layer sizes and learning rates required for convergence strongly depend on the type of temporal signal being encoded, depending on the experiment, different networks' organizations were devised via an iterative process. The process involved bottom-up, layer by layer, building of the network and assessing its performance. For each layer, firstly, the short memory temporal size was defined. Smaller for lower layers and progressively increasing for higher layers. Secondly, the layer was initialized with an over-sized number of nodes and trained for an artificially long training period. This guaranteed convergence and development of stable weights in long term memory. After that, the network was assessed by the output activation pattern or node weights, for the number of utilised nodes compared to total number of nodes. If the number of nodes activated was much smaller, then the total number of nodes in the map and their weight were clearly developed, the network size could be shrunk. After the optimal layer size was found, the artificially long learning period could also be reduced to the point where the overall final average Euclidean distance was still achieving relativity comparable value. The neighbourhood function reach usually followed the learning rate value such that the reach would recruit nodes from the whole map at the beginning but allow individuation of nodes at the end of the learning. Additional, initial neighbourhood reach was scaled in proportion to the map dimensions. After the layer's size and learning rate was defined the parameterisation of the next layer could commence. Since the network's layer can encode up to the certain time spectrum of sequence elements, the next layer's temporal view (short term memory length) can be usually doubled. In the end, the size of each layer depended on how many components at each layer's frequency spectrum the encoded signal included.

Chapter 3

Experiments

To test the hypothesis that the described architecture should be able to capture, learn and recall temporal characteristics of the signal, a number of experiments were devised. The experiments followed a progression of increasingly complicated signals being encoded by the network; starting from the simplest temporal signal (Experiment 3.1), they move through pure noise (Experiment 3.2) and more complicated synthetic signals (Experiment 3.3). This was later followed by tests checking network performance on dealing with signals buried in noise (Experiment 3.4) and real world data (Experiment 3.5). The experiments culminated in testing network's ability to capture and encode systems' temporal dynamics to aide discovery of hidden states in order to solve a simple Reinforcement Learning problem (Experiment 3.6).

3.1 Experiment 1: Encoding of simple repeating spikes

The objectives of this first experiment were to test the network performance with the simplest conceivable repeating signal and see if the signal could propagate through layers. Simultaneously, it would check if SOM connected to short term memory and with an additional temporal inhibition system would converge and to discover what the activation pattern and rate of nodes at each layer would be. The final aim of the experiment was to check whether the node weights after training could be used to decode the initial signal into a prediction. The main objective was to show that the new network could encode, remember and predict long dependencies of the simplest input signal. The maximum length of these dependencies was decided by the number of layers and length of input buffer queues in each layer. The network was tested on simple synthetic signals, for which it is easy to manipulate the temporal dependencies. The signal was a two-dimensional vector. The vector represented one-hot representation of one of two states. The signal has a simple temporal behaviour in which, for most of the time, one state is activated, but, at every defined number of time steps, activation briefly flips to the other state. This can be seen as a constant frequency spike in a discrete time domain. See the input signal in the figure 3.1. This kind of signal is relatively hard to predict, because the timing of the next spike depends only on the last visible spike, which is a small part of the overall input sequence. Manipulating the input signal frequency and the number of predicting layers finds the maximum delay, at which the network is no longer able to correctly predict the next input spike. To prove that this network could compress input data in time, the number of node activations was recorded on each layer. This shows the relationship between the activation on a given layer and the frequency of the input signal. To show that the algorithm converges, the average Euclidean distance between input and the BMU throughout the learning process was also plotted. Because convergence is dependent on the input signal, the first activation graph shows only exemplary behaviour for a single frequency, not a mean and standard deviation from multiple runs.

The parameters of the experiments were as follows: four SOMs of size 3 by 3 stacked into a 4-layer architecture. The input queue for each layer is a basic delay-line memory with increasing lengths of 2, 4, 8 and 16. The prediction window for each layer was the same as the size of the input queue. The learning rate for each layer followed an exponential decay and we used a Gaussian neighbourhood function. Each layer had a different starting point and the decay factor for learning rate allowed convergence of higher layers to take place later in the process. Propor-

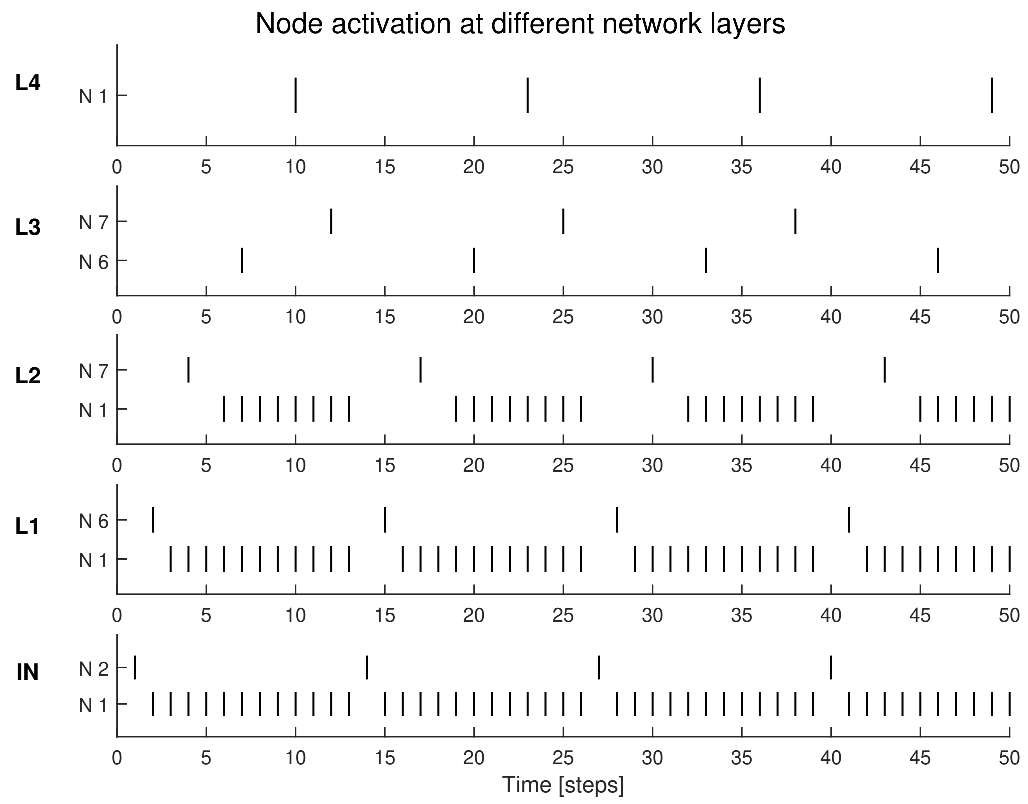


Figure 3.1: Plot of exemplary network node activation at each layer when subjected to the simple periodical input. Input signal frequency is 1/13.

tional learning rate for all layers was set to 0.5, while the exponential decay factors were -1,-3,-6 and -12 for layers 1,2,3 and 4, respectively. Temporal inhibition circuit settings were as follows: the global threshold was not allowed to drop below 0.5 for any layer. The persistence of the global threshold for each layer was set to be one time step smaller than input queue length. This pushed the nodes weights to not overlap when learning the input history. Prediction happened only on the top layer. The last node activated on this layer passes its predictions of the next node to be activated to the lower layer nodes. They are then combined with the activation on the lower layer and the sum is passed again on to the layer below, until the bottom input prediction is reached. For each frequency of input signal, the network was trained for 2000 time steps and then its prediction and activation was tested for 1000 steps. 50 runs of each input frequency were performed, and average and one standard deviation was calculated for each of plotted values.

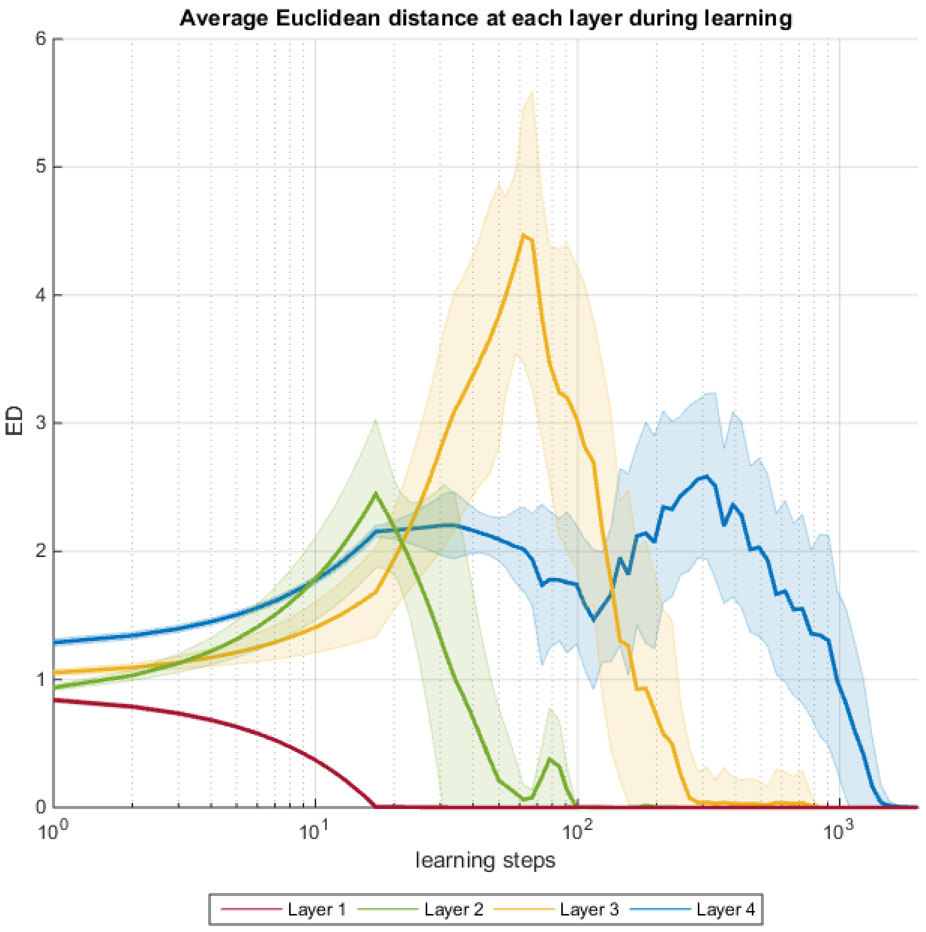


Figure 3.2: Average and standard deviation of euclidean distant between input vector and BMU node weights at each of the layers throughout the learning process. Input signal frequency is 1/13. The average is over 50 runs.

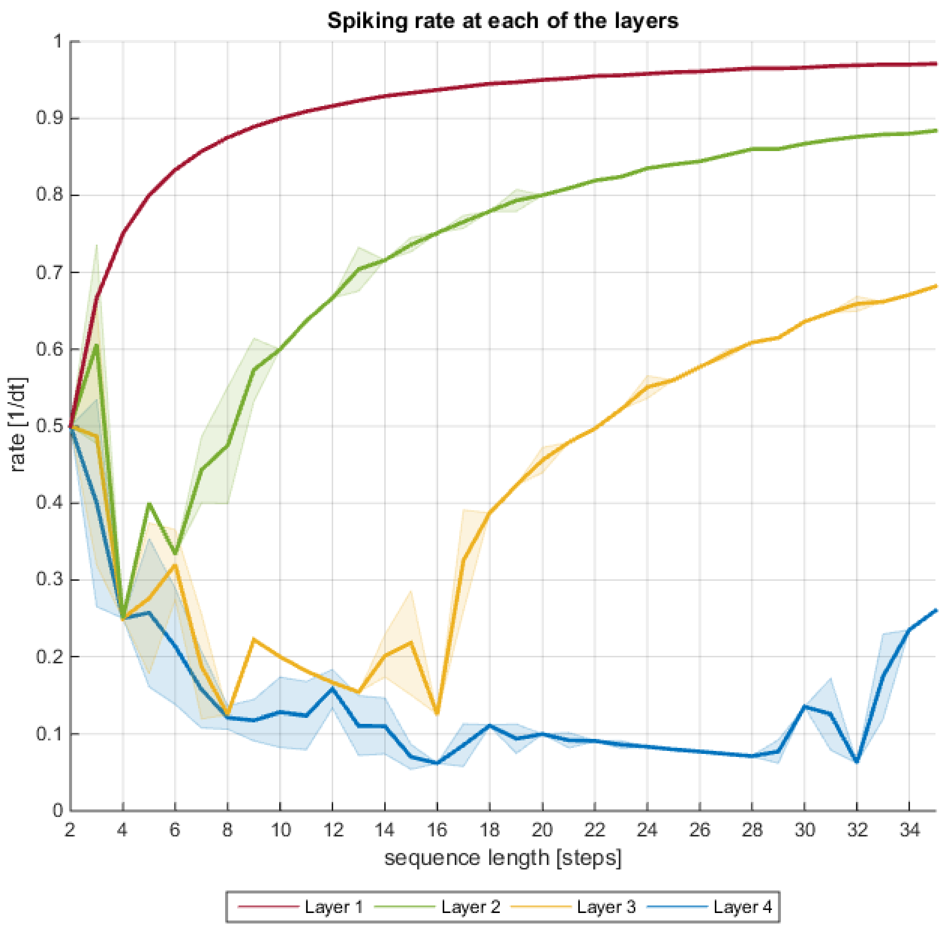


Figure 3.3: Spike activation rate at each layer over different sequence lengths (1/frequency) and predicting layer.

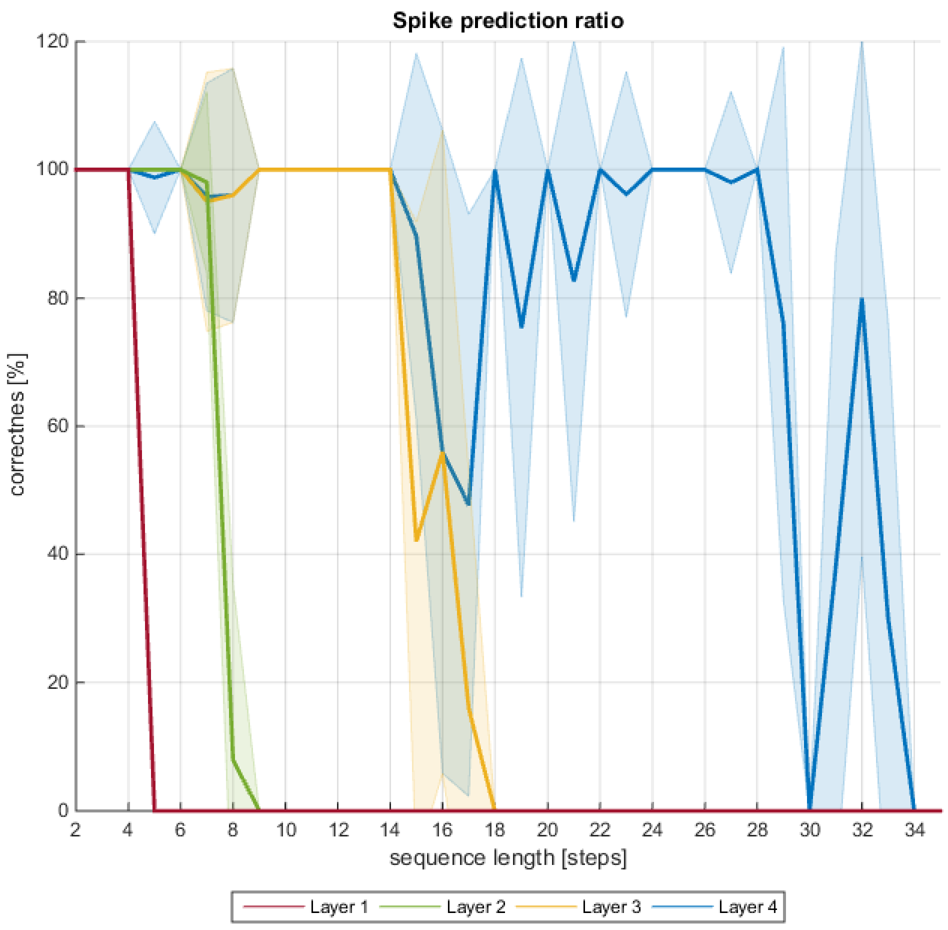


Figure 3.4: Average and standard deviation of spike prediction capability over different sequence lengths (1/frequency) and predicting layer.

3.1.1 Activation

First, the response of the network to the input signal was tested. The frequency of 1/13 was chosen because it engages all the layers simultaneously and does not stretch the network's capabilities to its maximum. Figure 3.1 shows an example activation of the network after the learning session. At the bottom we can see the input signal and above we can see the response of consecutive network layers to this signal. Only nodes that are activated on each layer are plotted. The rest of the nodes did not learn to respond to the signal. All layers learned to respond to specific periods of the input signal. The activation of nodes at the first and second layer does not explicitly correlate to the input timing. This is because of the repetitive activations of nodes at those layers. It can be seen that although the first three layers did encode the input signal, their queue length did not allow them to perform complete time compression. The fourth layer, with queue length of 16, was able to pick up the repetition of the input signal and learn its behaviour. This is also shown by the fact that this layer gets activated with the same frequency as that of the input signal and only needs one node to represent the whole sequence.

3.1.2 Convergence

In Figure 3.2, we can see the development of the average Euclidean distance between the input signal and the weights of the node declared as the BMU. Time is shown in logarithmic scale, because the learning at each layer takes a progressively longer time to converge. It follows an expected trajectory where the lower layers converge first. Additionally, it is only after one layer has converged that the next higher layer starts to converge. This can be explained by the observation that as one layer converges, it starts to find a more detailed description of input signal and the output gets more complex. As the output of one layer gets more complex, the Euclidean distance of the nodes on the next layer increases, because network is presented with previously unseen variability in the signal.

3.1.3 Activation rate

After testing the network performance with one frequency, experiments with sweeps through frequencies were conducted. The output of this experiment is shown in Figure 3.3. In this experiment, the network was presented with signals of different frequencies (1/sequence lengths). After the learning period, the node activation rate at different layers was measured while the network was tested with the same input signal. The node activation rate is defined here as a ratio of time steps when any node is activated over the total number of time steps. Figure 3.3 shows the average activation rate and standard deviation over 50 runs for each sequence length. The activation rate for each layer follows a frequency curve (1/sequence length) until it reaches roughly twice the queue length for that layer and then it starts to rise, not being able to resolve temporal dependencies.

3.1.4 Prediction

The results from the final experiment are shown in Figure 3.4. In this experiment, the network simultaneously learned the traditional node weights and also the additional prediction weights by way of associative memory augmentation. After the learning had converged, the chosen layer was used to predict the next input. Since each layer can only predict its own input, the higher layer's predictions were combined with lower layer's predictions, down to the network input layer. At longer sequence lengths, the opportunities for making wrong predictions were less frequent. This is caused by the less frequent changes in the input signal. To remove the effect of this artificially high base prediction correctness for longer sequences, the correctness was measured only for predicting the part of the sequence containing a spike. From Figure 3.4 we can see that the predictive power of each layer has a definite limit. This limit corresponds to the point at which the activation rate diverges from normal decay. Prediction from a next layer above almost doubles the prediction power of a lower layer.

3.1.5 Discussion

In this experiment the temporal inhibition circuit was still being developed and although the circuit would enforce refractory period and refractory level, the circuit would still allow the same node to be activated multiple times when the match was below the refractory threshold. It can be seen in the activation depicted in Figure 3.1, that the single node can be activated multiple times especially when the period of the signal is longer than the time horizon of the particular layer. It is even more visible in Figure 3.3, where at certain sequence lengths each layer's activation rate would spike up when the nodes started to be activated multiple times in the row at that layer. Since such intense activation of nodes is not informative and artificially increases the amount of spikes sent to the higher layer the additional rule for temporal inhibition circuit was added to stop that behaviour from happening. This rule did not allow the same node to be re-activated within the refractory period, even when the match would fall below refractory threshold. This rule was already described in the methods section, as all other experiments already employed this improved temporal inhibition circuit version.

3.1.6 Conclusions

The experiments showed that the network presented with a simple repeating signal could propagate this signal through layers. The SOM node weights can converge onto a representation in a sequenced, timely and predictable manner. The node activation rate is dependant on signal frequency and when this frequency rises above the layer capability to encode, the activation increases to allow the next layer to pick up longer dependencies. Finally, the converged weights can be read to recreate the signal in the form of prediction.

3.2 Experiment 2: Encoding of noise

Any network presented with real world data has to be able to discern between the actual signal and the associated noise. After showing in Experiment 3.1 that the network could encode noise free signal, in this experiment the network was tested as to how it would react to noise without any underlying signal and whether such signal would break any learning paradigms. Ideally, when the network cannot find any underlying signal dependencies, it should dedicate its resources to encoding all visible combinations of signal and re-represent them for the next layer. The next layer should then try to find longer possible signal dependencies. With a pure noise signal the combinational explosion of possible combinations means that the network resources are stretched to their limits and in this experiment would help to see how the proposed network dealt with reaching such limits.

This experiment was also conducted to test the capabilities of single layer to converge and re-represent signal without any temporal dependencies. The layer was trained on a randomly generated binary signal. The network should be able to converge a certain amount of its long-term memory weight in order to represent repeating sequence patterns. As the signal has no temporal dependencies, the weights should represent all possible signal combinations. The network should use all available nodes to represent the sequence, which should result in lowering the activation frequency of each node. Independent of signal type, the network should be able to encode the signal into node activation pattern, which should be reproducible into original signal.

The experiment was conducted on a single layer with constant short term tape-delay memory of 6 time steps. The network was trained on computer generated pseudo-random binary signal. Each training took 5000 steps for the network to converge and then network was tested for 1000 steps without weight updates. Network testing included finding the number of activated nodes, its capability to reproduce original signal from node activation and its weights, and determining the amount of total node activations. Additionally, after each training, the number of converge weight (above 0.9) was measured. Experiments were conducted with different map sizes to see how the algorithm utilises a different number of long term memory units.

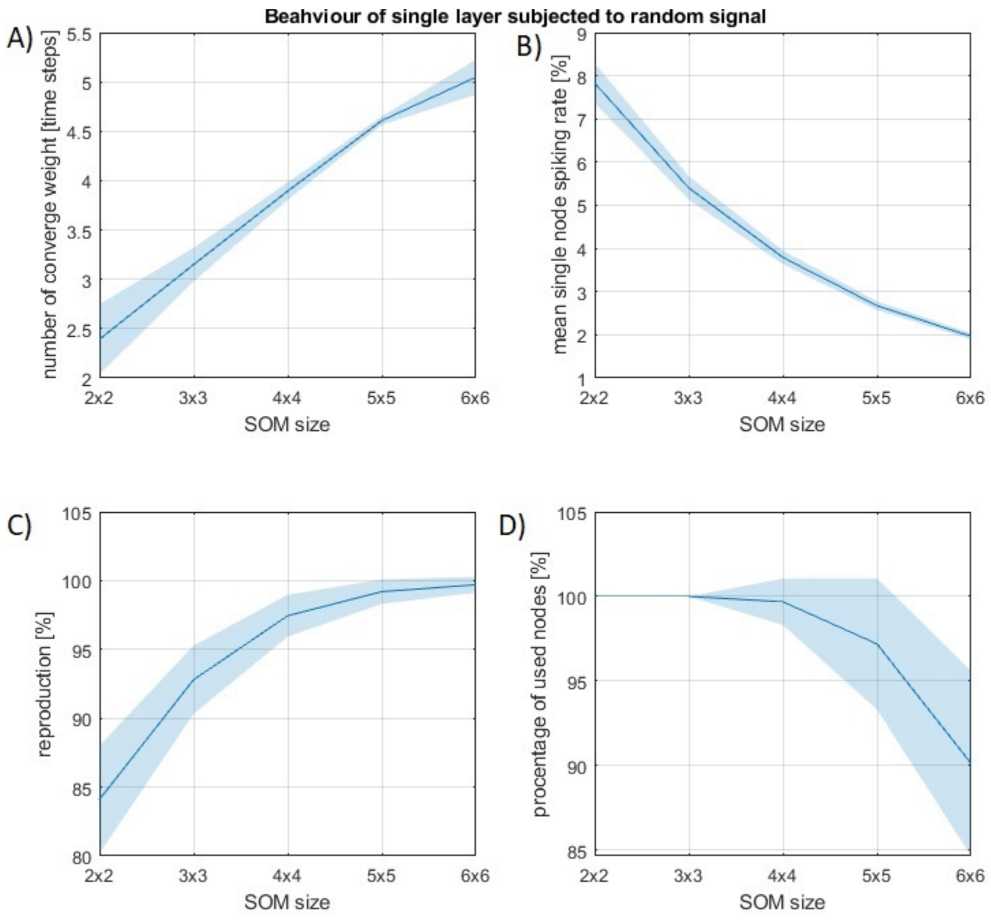


Figure 3.5: The figure shows statistical results of 100 runs of a single layer network trained on pseudo random binary signal with different layer map sizes. A) Number of converged weights on short-term memory at different map sizes. B) Rate of single node activation as a ratio of all time steps. C) Ability to reproduce base signal from node activation using learned weights. D) Percentage of nodes activated as a response to the random signal.

As the size of the network map increases, the number of converged weights increases see Figure 3.5A. The number of converged weight bits follows almost exactly logarithm of base 2 of the map node number, showing that the algorithm tries to encode all binary combinations of input signal in its weights. Simultaneously, the number of activations for each node in the map declines as weights start to represent bigger snippets of the sequence and do not have to be activated so often see Figure 3.5B. Figure 3.5C shows that the reproduction is not ideal for small map sizes, but quite quickly reaches almost perfect reproduction for any bigger map size. Figure 3.5D shows that the utilization of nodes starts to decline from maximum as map sizes increase.

This experiment shows that the presented network can work with signal that does not have any temporal dependencies within the layer's temporal perception spectrum and still encode, and, almost losslessly, re-represent and simplify signal for the next layer. The next layer can then try to find temporal dependencies in its longer temporal perception spectrum. The network does not produce perfect reproduction for small network sizes because of the algorithm's goal of spreading activation as far as possible and its not ideal influence of neighbourhood function. For bigger maps, however, the network utilization starts to degrade because of the problem of converging multiple weights in short training sessions.

The experiment also shows that the network performs as theorised: trying to employ the biggest number of nodes, shown in node utilisation, to represent the longest possible sequence snippets, in the form of the number of converged weights, and, simultaneously, still achieves good reproduction of the signal.

3.3 Experiment 3: Encoding of complex repeating signals

After the first two experiments 3.1 and 3.2 were conducted to test the very basic functionality of the network, the next experiment was conducted to challenge the network with a signal of a more complicated nature. A signal was chosen which had both short term and long term repeating patterns, which would mean that each layer of the network would have to find, encode and re-represent different characteristics of the signal. This experiment included study of feed forward activation patterns of nodes at different levels, a view of the weights representing long term memory and the possibility of mapping these representations into the original signal. The network was expected to pick up different characteristics of the signal at the different layers of the network, depending on the signal periodicity which was changed to see the changes in network behaviour.

3.3.1 Single convergence activation test

Since the network utilized an unsupervised learning method, each of the training sessions might result in different weights' convergence to the same input signal. This first test was only conducted to explain activation patterns; which is why it presents only one exemplary result. The input signal used in this test was a periodical sequence of 50 time steps representing the binary pulse density modulation of a sinusoidal wave created by a first order delta-sigma converter. A few periods of this sequence can be seen in the bottom of Figure 3.6IN as an input to the first layer. To keep constant length of the input vector, the sequence was augmented by its negation. Three layers were employed to encode this signal: two first with the SOM sizes of 3 by 3 and top layer of size 2 by 2 (see SOM of node weights in Figure 3.7A). The short term memory of the first layer was implemented as a delay line of length 4. Short term memories of the higher layers were implemented as a wave delay line with 8 memory units and speed of propagation of one unit per time step for second layer and 0.5 units per time step for top third layer. The network was trained for 5000 time steps with decaying learning and neighbourhood rates with higher levels decaying more slowly. The exact implementation can be found in Appendix D.

Figure 3.6 shows the resulting feed forward activation of trained network at different levels subjected to the periodical signal. The vertical lines represent activation of certain memory node. The bottom part shows periodical input signal presented to first layer. Figure 3.7A represents the weights of three layers of trained SOM maps. Each node weights represent a partial snapshot of lower layer activation. The X axis of each node weights image represents time and Y represent the different nodes from lower layer. It is possible to see a resemblance between the weights and activation shown in Figure 3.6. Figure 3.7B shows how weights from one of the top layer nodes can be de-constructed into the activation of a lower layer. Reconstructed activation of lower layer can then be reconstructed again

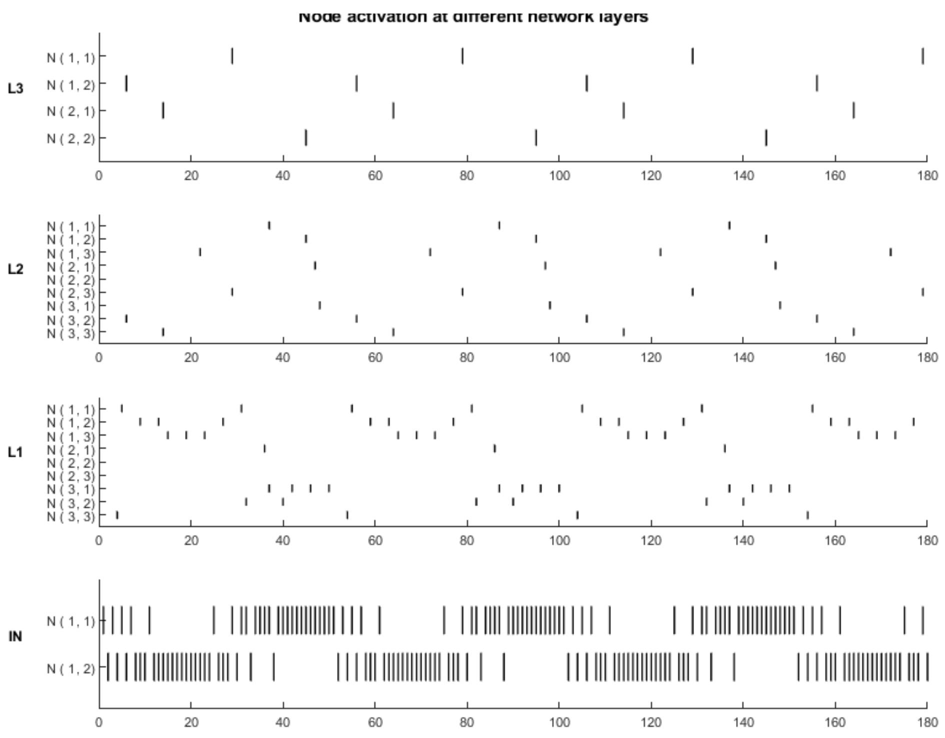


Figure 3.6: Node activation for a three-layer (L1 3x3, L2 3x3, L3 2x2) network with a two bit input vector. The network was trained on a density encoded sinusoidal curve with a period of 50 time steps and then presented with the same signal.

until the original input sequence is recreated.

The presented learning algorithm is capable of observing repeating temporal patterns and converging on representation encoding the pattern in internal long term memory weights. Each layer is capable of dividing the input sequence into smaller chunks and represent them in the long-term memory weights. Each layer captures repeating redundancy and simplifies the signal for the next layer. The SOM neighbourhood function creates a topology of encoded weights visible in the similarities in weights between neighbouring nodes.

3.3.2 Node activation dependent on sequence length

This experiment was performed to visualise where and how layers respond to different periods of input signal patterns. The experiment also checked whether the network could find and synchronise its activation to the input signal period.

The network of the same architecture as in the previous experiment was fed with signal generated by a delta-sigma generator. To change the pattern period, the

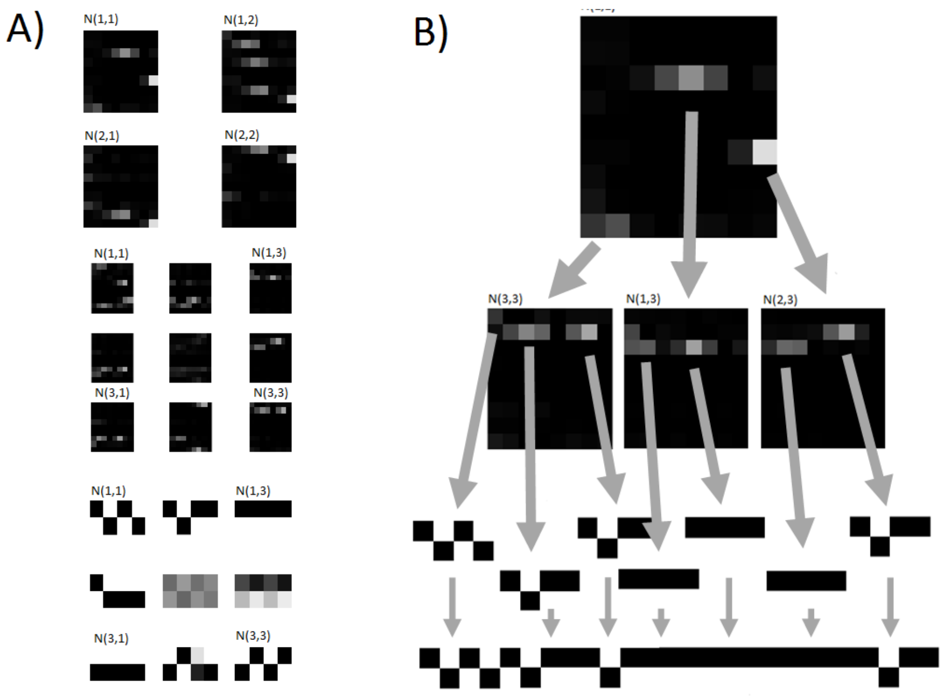


Figure 3.7: Weights of the network trained on a density encoded sinusoidal curve with a period of 50 time steps. A) SOM connection weights presented in 2D . Part of each layer activation at different layers can be decoded from each map weights image. B) Partial reconstruction of base signal input from activation of one of the top level nodes.

frequency of the fundamental source sinusoidal wave was changed at each run. After each training session the network was tested with the same signal, and the frequency of node activation was recorded. This created spectrograms of node frequency response for each input frequency. The results show average spectrograms of one hundred runs.

The spectrogram of node activation for the first layer in Figure 3.8 shows that node activation synchronises with the input signal up to certain limited value when the period falls out of scope of this layer and nodes start to be reused. Because of the nature of delta sigma modulation, signals of uneven period length fully repeat over double period length. Thus all uneven period values are met by a response of double period activation. This creates an activation spectrogram with two dotted lines. It can be seen from the figure that consecutive layers get activated at progressively longer signal periods. A layer activation start to synchronise with the input signal around the point when the period of this signal exceeds this layer's short term memory length. This synchronisation carries on for the rest of

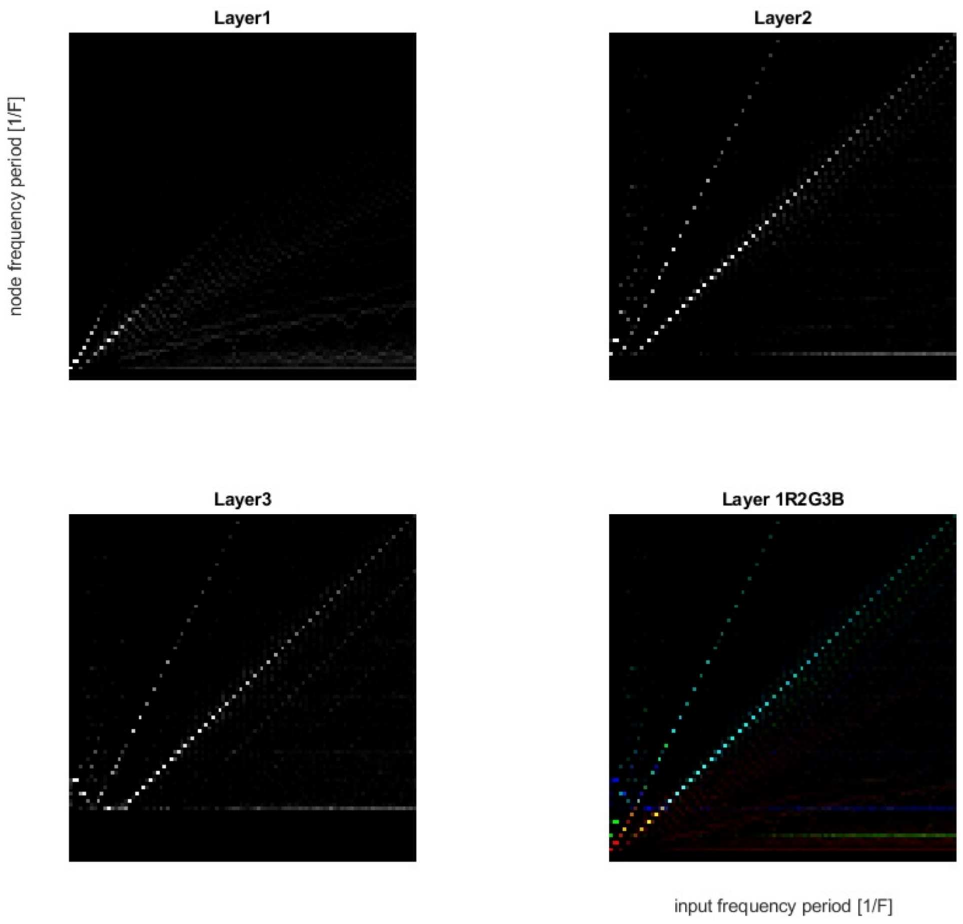


Figure 3.8: Spectrograms of nodes activation frequency when trained to respond to different input signal frequency. X and Y Scales represent sequence length not the frequency.

the signal periods. Similar behaviour could be seen in activation of all layers.

3.3.3 Network convergence dependent on sequence length

One of the main limiting factors while learning long sequences is convergence time. This experiment tried to test the characteristics of the network to converge on sequences of different length and determine whether longer sequences required additional iterations of the learning algorithm. There are a lot of parameters that might define single layer convergence time, like the number of nodes, sequence complication and sequence length with respect to layers temporal window. Yet, if we consider the worst case scenario we could imagine a sequence that exhausts all available layer nodes without any reuse and stretches node activation to maximum inhibition period. This situation may still have bounded iteration requirements not directly depending on sequence length, since one node has a limited number of activation that is required in order to converge its weight representation. Nonetheless, the whole network has to have enough layers to capture the sequence's whole temporal spectrum for signal to be fully encoded.

In this experiment, the network was presented with sequences of randomly generated binary symbols repeated to the network during training and testing. The sequence was generated with different fixed lengths. The network recognition, in terms of average Euclidean distance between activated node and input, was tested. Simultaneously, the number of presentations of the sequence during training is changed to find its effects on the final network performance. The network consisted of four layers, each with 3 by 3 map and progressively doubling temporal spectrum of short-term memory. Since this network is multilayer, each layer's learning is dependent on the lower layer's convergence. Since, it is difficult to isolate dependence when all layers are trained simultaneously, in this experiment the performance of each layer was tested independently. Thus, all testing and training for higher layers was done on output from lower layers' pre-converge in ideal conditions (200 sequence repetitions).

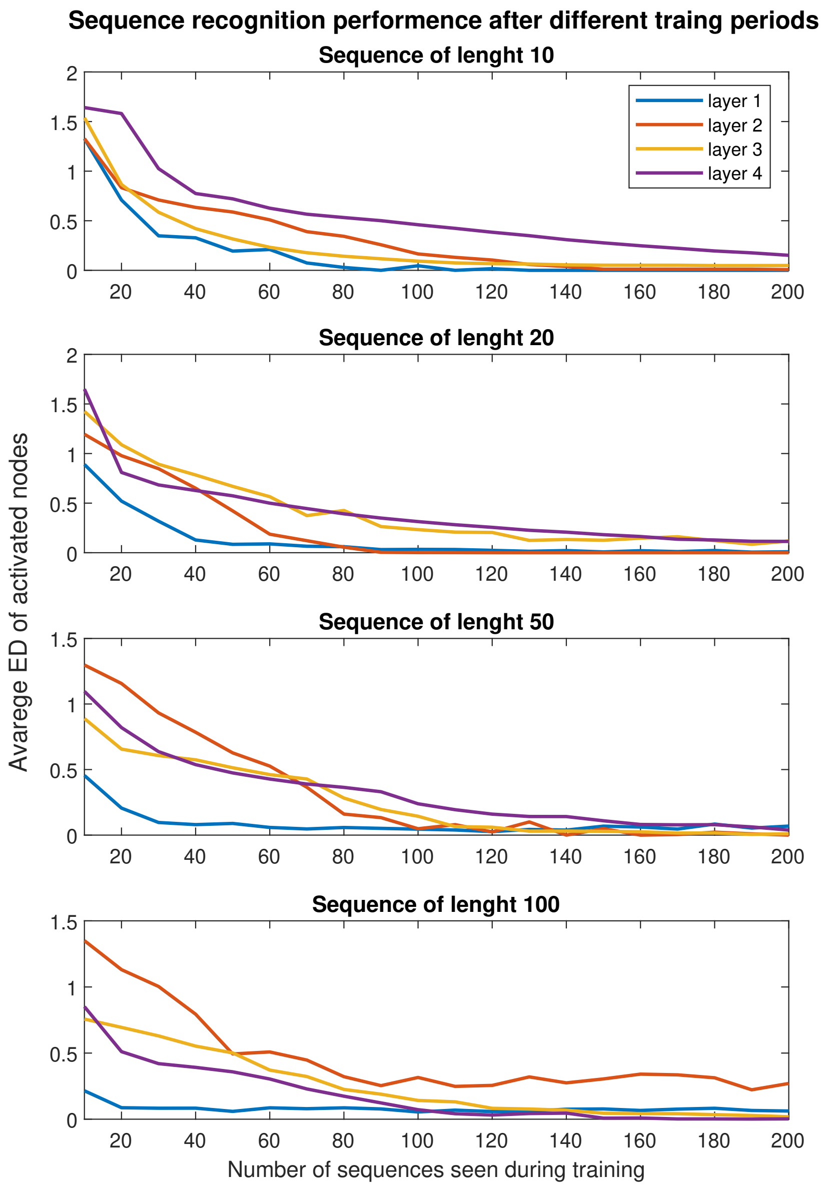


Figure 3.9: Network sequence recognition expressed as an average Euclidean distance of activated nodes in different training conditions.

Figure 3.9 shows the experiment results. It can be seen that layers reach their final performance around 100 repetitions of the given sequence, independently of sequence length. An exception to this is layer 4, which, with a shorter sequence length, takes longer to converge. This can be explained by the fact that its temporal spectrum is bigger than the sequence length, and it might be argued that it is not required for shorter sequences. Also, the second layer started to under-perform at the longest sequence case, which is caused by the nature of the signal, which at this time spectrum starts to overwhelm the number of nodes at this layer.

It was shown that within the spectrum of constructed network, the number of sequence repetitions required for convergence does not grow with sequence length. An increase of sequence length requires more iterations of learning algorithm, but it is fully justified by the increase in number of weights required to be converged. To be able to increase the capability of encoding to even longer sequences than tested here, additional layers would need to be added to the network. Since each additional layer can multiply temporal spectrum, computational expense of using this method for encoding sequences only grows logarithmically to the sequence length.

3.3.4 Conclusions

The network is able to synchronise to the input signal and shift the responsibility of synchronisation between layers, depending on signal period. The network performs frequency decomposition in some way similar to the fast Fourier transform (Heideman et al., 1984), where signals are progressively decomposed based on their frequency components.

3.4 Experiment 4: Encoding of signals with noise

All real world signals are a combination of the underlying signal and noise. To respond to such real world data the network has to be able to separate the underlying signal from within the noise, both during training and testing. To have that ability, the network has to be able to train on noisy data and then be able to recognise the underlying signal within it.

Since it is difficult to find data sets with signals of changing noise content, the signal used in this experiment was artificially constructed to gradually challenge the network with an increasingly noisy signals to ascertain its limits. Experiments were conducted on a three-layer network with the same hyper-parameters as the previous experiments, trained on the same signal generated by a delta sigma modulator with a period of 50 time steps. The Euclidean distance between the input and the connections was used to indicate the encoding error during testing. This error defines the mismatch between the tested signal and network response. This mismatch can be recognised as the measure of the network's ability to recognize the signal.

3.4.1 Spatial noise

First, the effects of spatial noise were evaluated, introduced by randomly flipping sequence bits. Separate networks were trained on noiseless and noisy (5%) signals. The response was then tested with signals containing different levels of noise. This type of noise reaches its maximum value at 50% since at higher levels the signal starts to resemble its negation which is interpreted as the same signal but shifted in time.

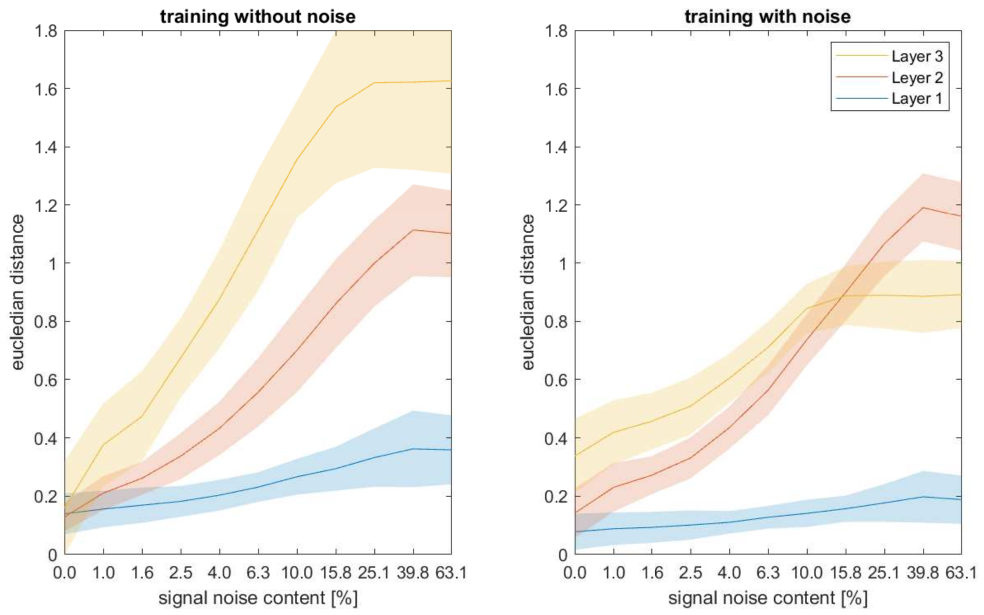


Figure 3.10: This figure presents a network's ability to recognize signal and generalise as noise in spatial dimension is introduced to the input signal. It compares average Euclidian distance when matching input and weights of activated nodes as noise is introduced to the input signal. The left plot shows average Euclidean distance when network is tested with noisy signal after training on noiseless signal. The right plot shows average Euclidean distance as network is tested with noisy signal after training on signal with 5% spatial noise.

In the case of networks trained without the noise, the error almost reaches zero when testing on noiseless signal Figure 3.10Left. This is expected since the test and training signal are identical. As the noise is increased in the test signal, the error starts to increase. The increase is slight in the first layer since it cannot differentiate between signal and noise patterns. The increase starts to be visible for the second and even more so, for the third layer. The error increases until the noise level reaches roughly 25% after which there is little visible difference for further increases in noise. When the network is trained with 5% spatial noise the error on a noiseless signal is lowest, as expected, but does not reach zero, Figure 3.10Right. It can be seen that the first layer of such networks perform better than in the first case, because it learns to encode all signal combinations and becomes less pattern specific. The second layer performs almost the same as in the previous case, except for a small reduction in error for noise levels similar to the ones used for training. The mismatch for the third layer is also smallest at the zero noise level and rises as the noise is introduced and plateaus around 10% mark.

3.4.2 Temporal noise

Secondly, we evaluated the effects of temporal noise during training. This noise, called *jitter* was set to randomly extend and shorten the signal by one time steps in a given amount of randomly chosen time steps. Separate networks were trained on noiseless signals and signal with 2% jitter. This amount of jitter results, on average, in one time step every period (1/50) being extended or shortened by one step. Network selectivity was tested by presenting the trained networks with the signal generated by the same delta sigma modulator as above, but with different periods. Therefore the network was trained on signal with varying, "jittery" frequency and then the network frequency selectivity was tested.

In case of temporal noise, when a network is trained without noise, its error is equally high on all testing patterns except the one trained on as shown in Figure 3.11Left. When jitter is introduced during training, the error is still lowest on the noise-free signal, but it also achieves a relatively low error on signals with similar period, e.g., 1/49 and 1/51 see Figure 3.10Right. It can be seen that as jitter is introduced during training, as expected, the selectivity of the network in testing is widened.

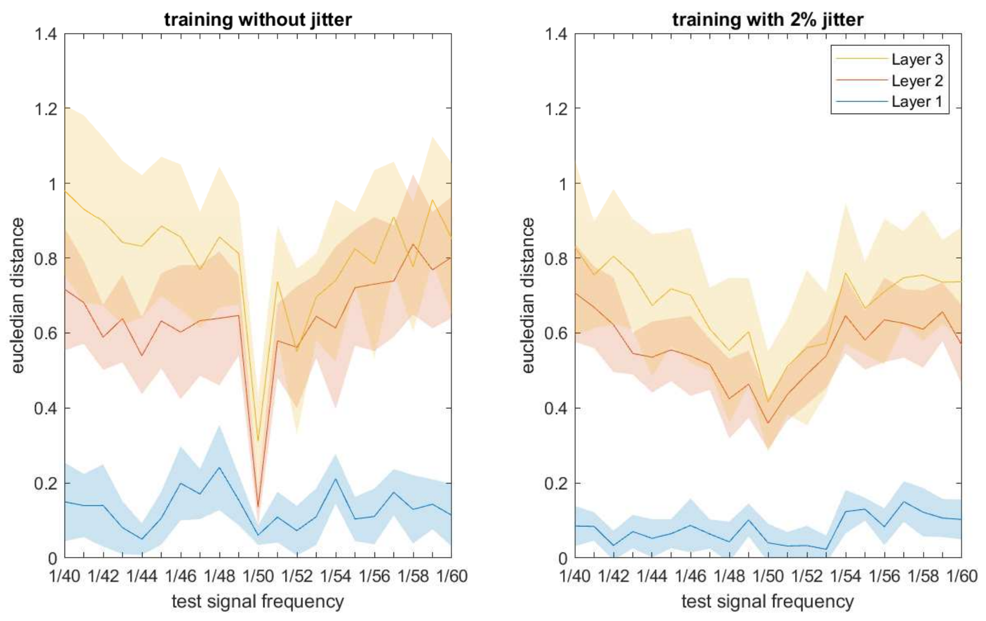


Figure 3.11: This figure presents network ability to recognise signal and generalise as noise in temporal (jitter) dimension is introduced to the input signal. The left plot shows average Euclidian distance when network tested with different base frequency signals after training on noiseless signal. The right plot shows an average Euclidian distance as network is tested with different base frequency signals after training with 2% of temporal noise (jitter).

3.4.3 Conclusions

The network can recognise signal within noise both during training and testing. The gradual introduction of noise in the testing signal creates a clear decrease in the network's ability to recognize the signal. The recognition is still visible even for quite high noise level (25%) in spatial dimension. When the network is trained on a signal with temporal noise, it has a low error on the original signal, but is also able to generalize its representation to lower its error on similar signals with slightly different temporal periods. Coping with spatial and temporal noise is made possible by the higher layers' capability to represent more abstract signal elements. The experiments showed that both in cases of spatial and temporal noise, the network can be trained on noisy signal and generalise to find underlying signal, shown by responding most strongly to noiseless signal during testing. That is the most crucial attribute of any neural network and this network not only managed to generalise spatial noise but also was able to generalise temporal noise which most of temporal network architectures struggle with. Coping with such noise can be difficult, especially for single layer temporal networks like LSTM or HTM see Y. Cui (2015).

3.5 Experiment 5: Encoding for classification

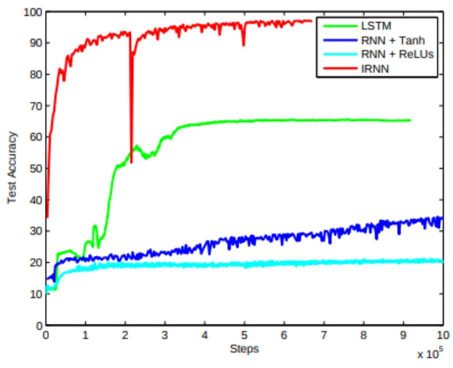
After the network performance was proven on synthetic data, the next step was to challenge it with a large dataset of real-world data. The experiment was designed to show whether a simple network could be trained in an unsupervised manner on a huge dataset with multiple possible sequences interlaced, while simultaneously showing whether the SOM could be used in a classification task just at the top layer of the network. In principal, the network should be able to encode, converge and re-represent even the complicated real world date without a huge loss of information, and correct classification should be a good proxy for that.

In order to compare the sequence encoding capabilities of the network with standard temporal supervised models, a top classification SOM was added to the network and was trained according to the algorithm presented in the Methods section and later trained on the well known sequential-MNIST (Lecun et al., 1998) dataset. The input to the additional top classification layer was the network top layer activation pattern mapped into wave delay line. The network had three layers in addition to the classification layer. The first layer was used as a thresholding function with only two nodes and no short term memory. The second layer consisted of a 3 by 3 map with a wave delay line of length four at its input. The final layer also had a 3 by 3 node map and a wave delay line length of eight units. The classifier layer was of size 15 by 15 with inputs connected to a delay line of length 392 units mapping history of 784 time steps. As per the classification algorithm, during training, the input of the top level SOM layer, was augmented by the class identity of the current sequence as a one-hot vector. By adding class identity to the input, the topological mechanisms of SOM grouped patterns of the same class together. Weight adjustments for this layer took place only at the end of each digit sequence presentation.

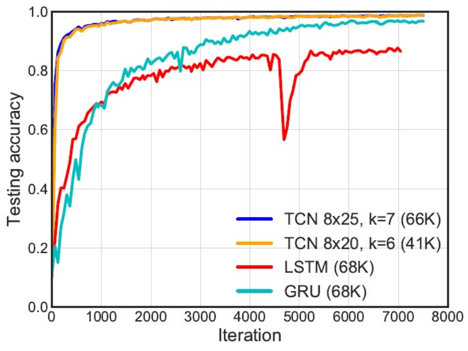
During classification, the input vector with unknown class identity was compared to all SOM nodes and prediction was made from the closest nodes, which had well developed class identity weights (>.75). This additional requirement stopped predictions from nodes lying on the border of class regions. This type of classifier is comparable to a k-nearest-neighbours' representation with the fixed number of neighbour skeletons in the form of the node weights. All layers were run through one epoch and tested on a 10k test set. The results are presented in Figure3.12.The network converged on representations, which could represent signal in the form that still consisted of most of the important classification information. The final classification accuracy was lower than large LSTM the accuracy of large (70K) LSTM networks, but better than smaller (10K) networks. The final network performance depended on the training period and the network size. These are non-trivial to compare as the networks use their connections differently. The network had 772 weights in the lower layers and 794K weights for the final classification layer, but both the sequence encoding and classification layers performed weight updates at a much lower frequency than the LSTM.

The main benefit of using this network compared to other algorithms, is that the layers of the network, except for the top classification map, work in an unsupervised manner. This allow the network to acquire the model of the sequence purely from the data. The acquired model allows the sequence to be filtered and compressed before being presented in simplified form to the classification layer. Additionally, the lower layers store the representation of the sequence, so that the classification network training only has to take place only at the end of each sequence presentation. Although the number of weight incorporated in the network is large, only a small amount can get adjusted in each update step and big classification network gets updated only after each sequence presentation.

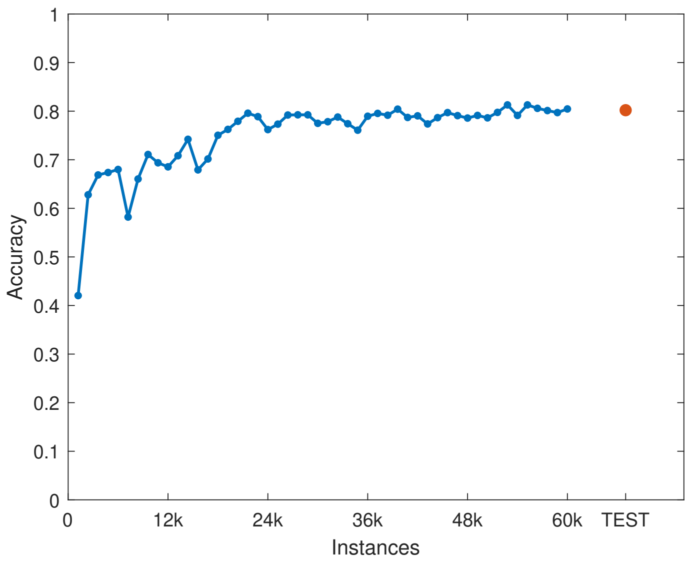
The experiment shows that the network can be trained in the unsupervised manner on a large set of data with a number of sequences interlaced, and encode, converge and re-represent the information in the format, which allows for classification comparable with other supervised temporal networks.



(a) From Le et al. (2015)



(b) From Bai et al. (2018)



(c) DMN performance

Figure 3.12: Sequence encoder performances on sequential MNIST. Sub-figure 3.12a shows that RNNs (10K parameters, accuracy 0.98) initialized using an identify matrix outperforms an LSTM network (10K, 0.65). Sub-figure 3.12b shows how a TCN (70K, 0.99) outperforms an LSTM network (70K, 0.87). Sub-figure 3.12c shows that ours network performance (0.80) falls between the performances of the 10K and 70K LSTM networks.

The exact implementation for this experiment can be found in Appendix E.

3.6 Experiment 6: Encoding for reinforcement learning

In this experiment, the new neural network architecture was employed in solving simple Reinforcement Learning problems. In this set of tests, the network would be challenged to encode a noisy signal sequence like previously, but also to actually take part in the active development of the sequence. This is because the developing representation of the sequence inside network weights would be used to guide the actions selection which in turn would change the perceived sequence. The network would be used to encode the state-action-reward dynamics of the RL problem, while the problem will be actively solved by prying those dynamics from the network. The network, in itself, would be a part of a closed-loop system trying to develop a model of the interaction. Although all the encoding of the signal would still be done in an unsupervised manner the close loop system in which the network will be employed would operate in an Reinforcement Learning manner. It was unknown whether the unsupervised learning mechanisms in the network were capable of converging to one stable representation and would be able to adapt and re-develop as the representation changes while still maintaining enough valid past representation to guide gained action selection knowledge.

The problem attempted was a simple toy scenario of a robot exploring a maze. The robot moved between maze grid cells by taking north, east, south and west actions in discrete time steps. The robot had limited sensory perception, allowing it only to observe whether the cell next to the current position was open or closed, making the problem partially observable. When the cell at the direction of action was open, the robots moved to it, but if it was closed the robot remained in the current cell. The maze was a simple corridor of cells ending with a T junction. An illustration of the problem can be seen in Figure 3.13. The robot always started at the most northerly end and received an initial crucial observation. This observation informed the robot at which end of the T junction it should finish the episode in order to receive a positive reward. When the robot moved to the wrong end, it finished the episode without the reward. The correct direction transmitted in initial observation was randomly selected at the beginning of each episode. This example problem was first formulated under the name of "tiger problem" since the reward or penalty was supposed to represent a hidden tiger in one of the cells at the end of the corridor. This problem was developed to test algorithms' ability to bridge the time lags between crucial (pointing to reward) observation and crucial (taking right turn) action. One of the features was the ability to extend the length of the corridor to increase the problem's difficulty. Additionally, the corridors following the T junction could be extended to increase the time lag between crucial (taking right turn) action and the final reward.

As discussed in the Methods chapter, the type of RL algorithm used in this experiment for solving this problem is referred to as feed-up (FU). Its characteristics are similar to some Dynamic Programming methods. The feed-up algorithm proposed here assumes that the reward will be received in the next time step and "asks" SOM network which nodes are consistent with such an outcome by means of SOM associative encoding for classification. This allowed the algorithm to determine the amount of association of receiving a reward to each network node, which can be interpreted as state-action value. An analogical algorithm was used to take the current short term memory and find nodes that represented the trace most closely resembling the current situation encoded in delay lines. Then, for each node on the top layer, the state-action value and resemblance to the past value were multiplied and the node with the biggest product was assumed to represent the most desirable future. This node then dictated the action taken at the next time step.

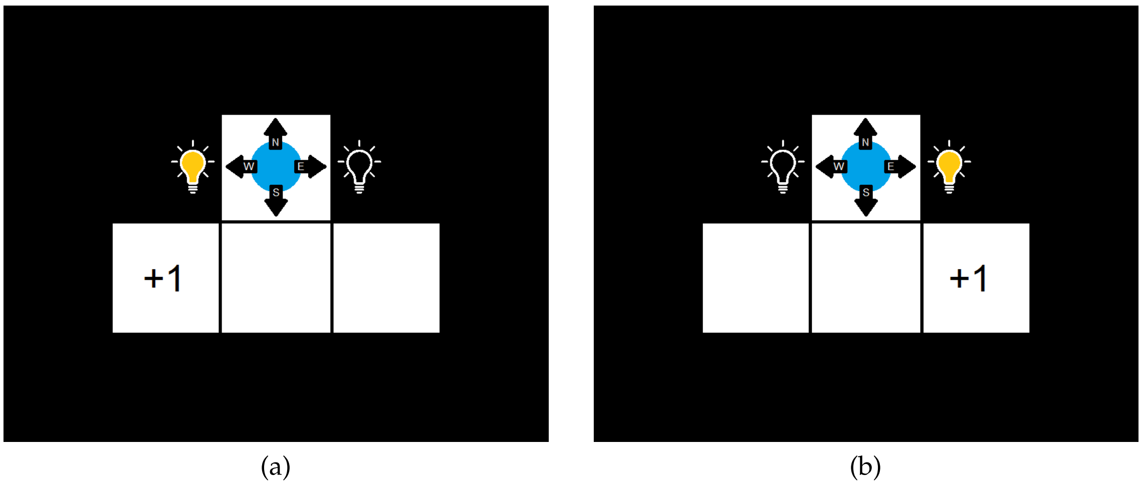


Figure 3.13: Visualisation of a simple reinforcement learning problem. It is a variation of the well known tiger problem []. The robot navigates this simple grid world taking north, south, east and west actions at every time step. It starts at the top of the map where it receives crucial observation (light bulb) that informs where the reward or penalty (tiger) is hidden at the end of the corridor. The placement of the reward (left or right room) is randomly selected at each new episode and the algorithm has to learn to connect the initial observation to action taken at the T junction. The length of the corridor before and after the T junction can be extended to test network memory retention.

3.6.1 Two step "Tiger Problem"

In the first experiment the simplest version of the "Tiger Problem" was approached. The maze shown in Figure 3.13 was used. It has only two steps required to pass through from start to finish. The robot would receive the crucial observation at the start and take one step towards the junction. It would then make a decision as to which way to turn to reach the reward. As in any RL problem, additional exploration rate was introduced, so that 10% of actions were randomly decided. This allows exploration of the problem at the start but increases the average number of steps required to reach the reward at the end of convergence. A two layer network was used. The first layer fed with observation, action and reward triples did not have any time dependence and was to compress the number of all possible observation, action, reward combinations. The second layer had a simple delay-line short-term memory of size 4 time steps. Which was sufficient to solve the problem even with the additional exploration steps. To allow the problem to become continuous, multiple episodes were lined after each other. To stop the network from joining rewards and short term memory between episodes, additional 4 clearing steps are inserted between episodes to clear the network. The example of activation of converged network solving the maze problem can be seen in Figure 3.14.

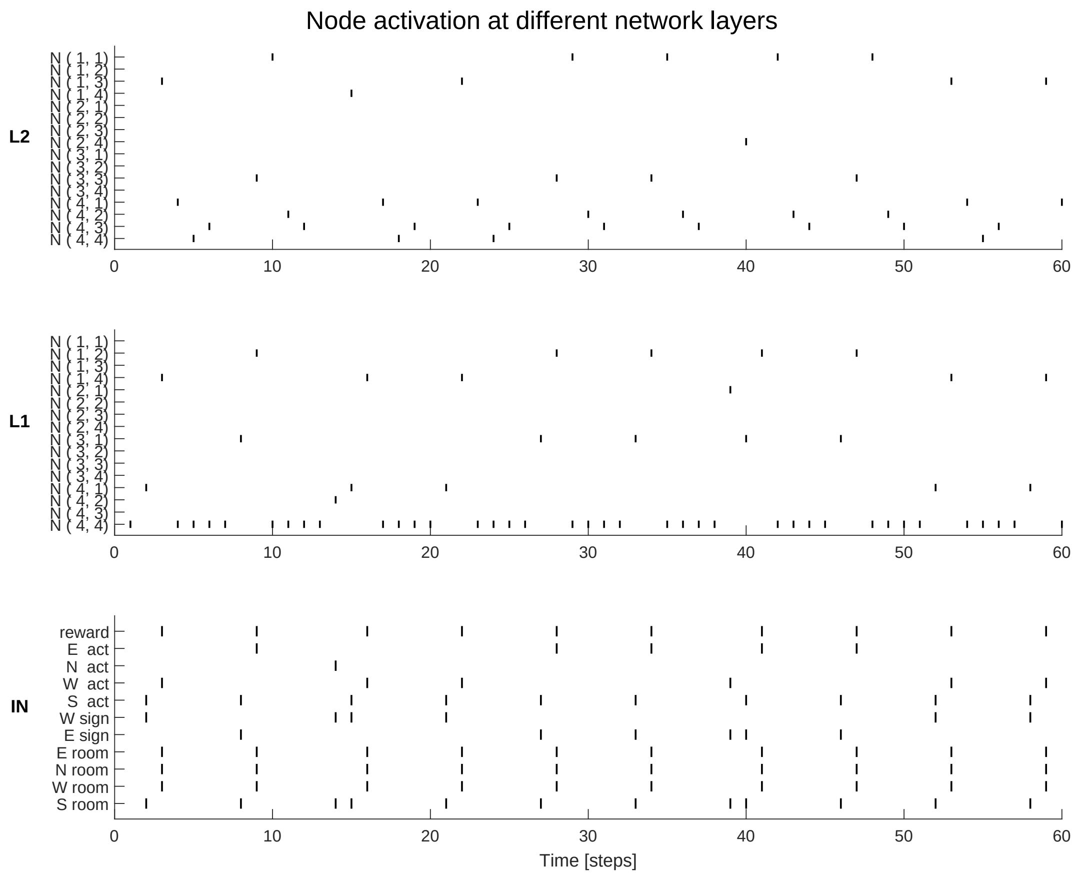


Figure 3.14: Example activation of two layer network solving simple Tiger problem. The input consist of 11 elements: a reward, 4 actions and 6 elements of observation.

A network, with two SOM of size 4x4 at first and second layer, was trained for 20000 time steps with decaying learning and neighbourhood rates to achieve convergence. While converging, the network was also involved in solving the presented RL problem. By creating the representation of successful traces in its node weights it allowed almost perfect performance in solving the RL problem. The average convergence of such a network can be seen in Figure 3.15. The experiment shows that average number of steps between receiving the reward is around 7 steps. Those include 4 clearing steps, 2 steps to traverse the maze and a sporadic exploration step. It is worth noting that when the random exploration steps fell on the decision step it could force the network to skip reward in this episode significantly extending the number of steps between rewards. The convergence experiment shows that this type of network is capable of reliably learning to solve the presented toy problem.

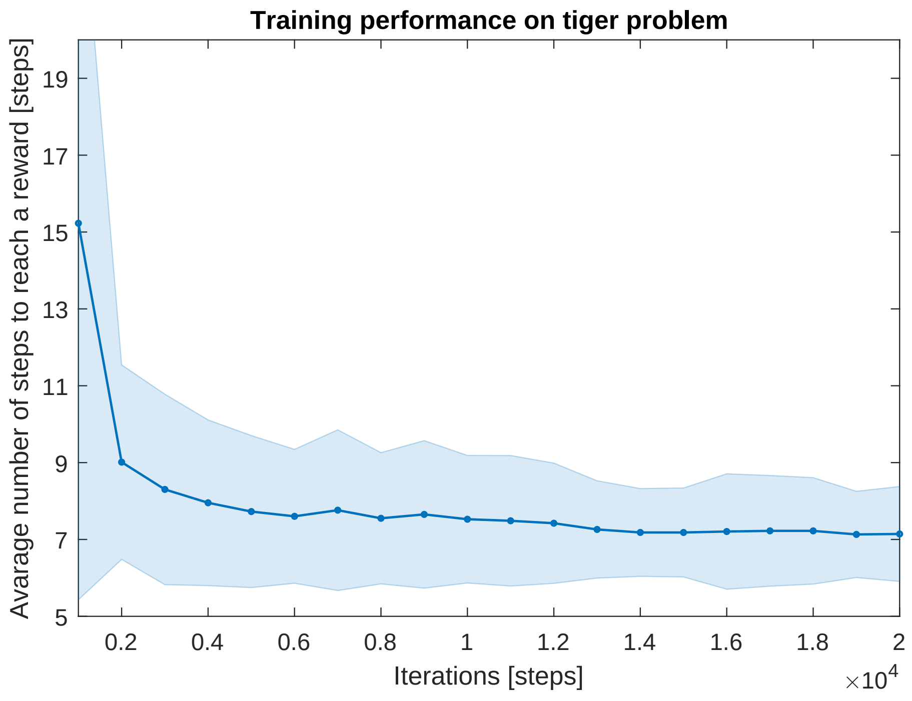


Figure 3.15: Convergence of network performance on the simple tiger problem. Optimal navigation through problem takes 2 steps but additional 4 clearing steps are added after every episode. Average and standard deviation of 100 runs.

3.6.2 Delayed reward updates

Since the reward signal is encoded in slowly changing network weights, it does not allow the network to change RL decision very swiftly. Thus, the calculation of mapping the encoded rewards into state-action values of the network nodes does not need to be updated after each network time step. To explore the implications of slowing down this reward association the further experiment was devised. In this experiment the reward association to network nodes state-action values is intentionally slowed down. By performing these calculations every so many time steps, the loop of interaction between the network and the RL problem was slowed down. The experiment setup for the network was the same as for the previous experiment, except the forestalling of reward updates. The results can be seen in Figure 3.16. After choosing forestall value from logarithmically increasing set, the network was converged 20 times and average progression of convergence was recorded. The results show that when the reward updates are done at least every 20-50 time steps, the convergence is similar to that shown when updates happen every time step. Even when reward updates are as rare as every 500 steps, the network converges slower but still to the same optimal solution. When network reward updates are less often then every 500 steps the network converges, but final performance is sub-optimal. From the results it can be concluded that the reward updates have to happen with similar intensity as the actual reward signal. At the first 5% of convergence the average number of time steps between receiving the reward oscillate around 14 steps. It can be argued that there is no need to perform reward updates with higher rates since they do not reflect any underlying changes in reward encoding. The results of this experiment show clearly that performing reward updates at a higher rate then every 20 steps does not provide any additional gain in speed of convergence. When the whole convergence cycle take 20000 steps, it is understandable that convergence performance degrades when reward updates happen rarer then 1000 steps. This reflects only 20 updates during the whole convergence and evidently is not enough to keep up the feedback of exploring the problem and keeping up with improvement of policy.

3.6.3 Extended "Tiger Problem"

In the last experiment, the difficulty of the RL problem was expanded. As mentioned earlier, the corridors of the tiger problem can be extended to require RL algorithm to bridge bigger temporal gaps. The first extension was the delay between crucial observation and deciding action. The second was the delay between deciding action and reward signal. The Feed-Up algorithm should be able to solve the first extension but it was incapable of solving the second extension since it did not try to anticipate rewards in future time steps. Additionally, a second class of algorithm was tested wherein the estimation of network node value was not

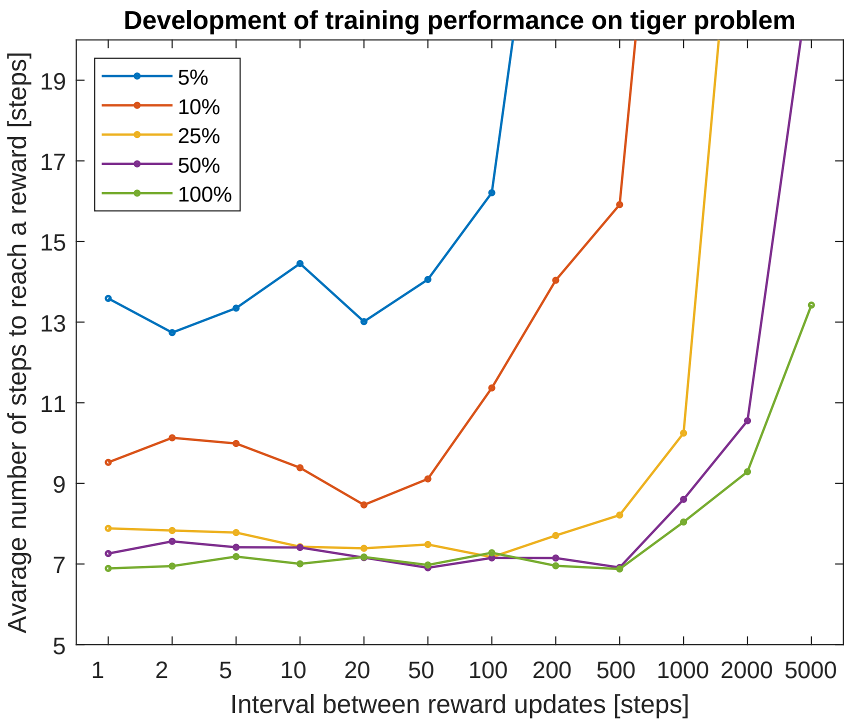


Figure 3.16: Convergence of network performance on the simple Tiger Problem with forestalled reward updates to higher levels. Average over 20 runs

based on rewards encoded in weights, but rather was purely based on temporal-difference calculations derived from nodes activation and reward signal. In this experiment, three scenarios of RL problem map were tested. For each problem map the same network configuration was used as in previous experiments; the only difference being the source of network node values calculation. Node values here were calculated either based on the previous Feed-Up (FU) algorithm or based on Temporally-difference Learning (TL). Results of the experiment are shown in Figure 3.17.

For the simplest problem map (1+1 Grid) the FU algorithm shows slightly better performance than the TL algorithm. For the problem where memory retention between the crucial observation and action was extended (2+1 Grid) both algorithms still solve the problem, but the TL algorithm had a significant advantages. When memory retention was extended between the crucial action and reward (1+2 Grid) the FU algorithm fell back to a suboptimal solution which was twice as long (16 steps), matching the performance of a algorithm ignoring initial observation. Conversely, the TL algorithm could still solve the delayed reward problem (1+2

Grid) showing slightly worse than average performance at the extended decision problem (1+2 Grid).

The results show that both algorithms produce node value estimates that can produce algorithm convergence and lead to finding solutions to RL problems. The TL algorithm, although simpler computationally, can outperform the FU algorithm in the presented extended problems. The experiments expose the shortcomings of current implementation of the FU algorithm leaving open the possibility of implementation of more detailed search for future rewards in network nodes. It also leaves speculation as to how such algorithms would behave and interact in multilayer architecture, and whether there is a possibility of combining value estimates from both methods for complementary benefits.

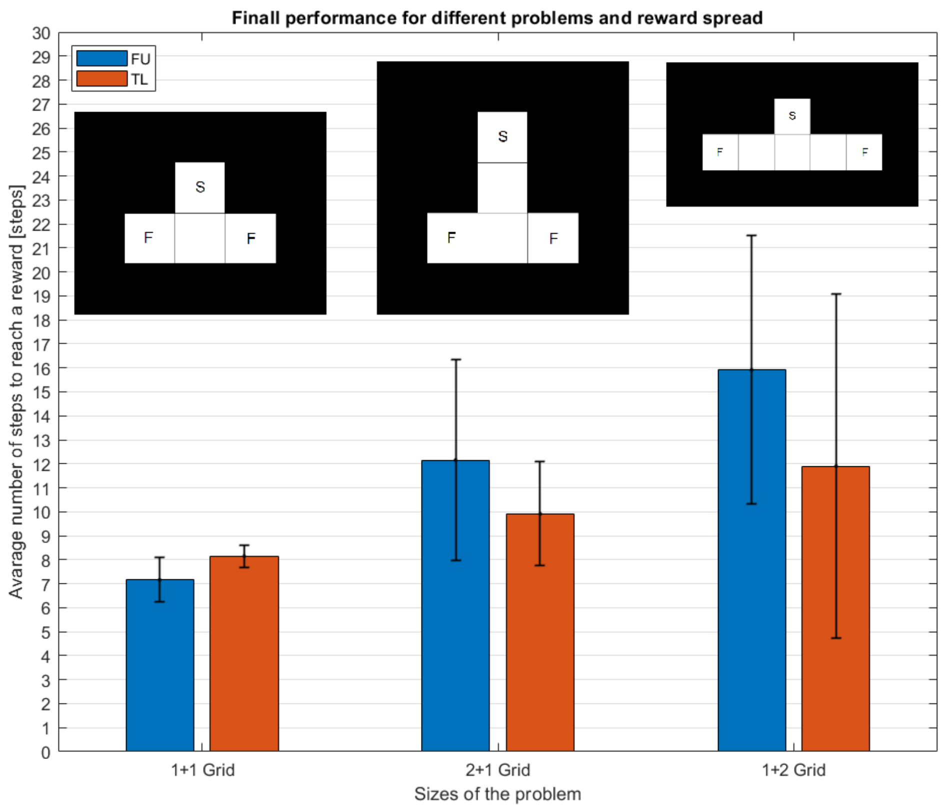


Figure 3.17: Convergence of network performance on different tiger problems and node value estimation mechanisms. Average and standard deviation over 100 runs

3.6.4 Conclusions

The presented combination of unsupervised model learning in a closed loop with RL algorithms can solve the most basic POMDP toy problems. In such a system the network takes an active role in developing the understanding of the problem dynamics. This feature allows for usage of such a networks in solving simple RL problems.

Chapter 4

Discussion and Further Work

This work only scratches the surface of developing a new neural network algorithm. Although the ability to encode different classes of signals was shown, the applicability to real world problems and in-depth comparison to other competing architectures was far from established. From this point multiple directions of development are possible.

One could aim to improve theoretical understanding of such a networks, which would involve deep exploration into all network's hyper-parameters and how they influence convergence speed, memory capacity, resistance to noise etc. For such exploration, a synthetic full defined signal would have to be used. Partial exploration into these questions has been done here, but in-depth systematic exploration would be required.

A second direction of development should be to establish each part of the network as the most suitable for its purpose. The network presented here was constructed from parts of other network architectures. This allowed for quicker development, since each part has an established history of performing certain tasks, but at the same time was not exactly developed for the specific role that it is used for here. If we take the Self-Organising Map as an example, generic neighbourhood function and calculations of best matching unit based on Euclidean distance might not give the best performance. In fact, there might be the possibility of developing some a more refined neighbourhood function taking time dimension into consideration, in order to make more optimal utilisation of node weights. There is even a possibility of redesigning the whole SOM architecture into a network capable of producing sparse encoding of signal, similar to HTM (Y. Cui, 2015). By representing the signal by activating multiple nodes instead of one hot best matching unit, the network could automatically increase representation capacity. This would involve redesigning SOM neighbourhood behaviour, allowing a split into few localised centres. This research for possible alternatives to best matching unit technique could create sparse encoding allowing a greater possibility of information compression. The question also arises whether this combination could open the possibility of multiple best matching units describing segregation and the combination of signals.

A third direction of development could be a comparison of a scaled-up version of this network to other established temporal encoding neural networks on real world dataset classification problems. One example of such a comparison was presented here, but the estimation of speed of convergence and resources required by each network were only partial.

A fourth area of development is in the reinforcement learning context. Again, in RL context the theoretical and practical directions can be taken. The research presented only used simplified implementations of RL algorithms. Since the network in fact creates models of interaction with the environment, there is a possibility of creating an algorithm, that is in effect intelligently searching through all possible developments of future and calculating confidence, risk and reward for following certain policies. Since the network is only an observer of the interaction with the environment, there is a possibility of having multiple networks creating their models simultaneously. This could allow for the constant development of better models by young networks, while the converged (old) one would provide required actions. This type of bootstrapping could even be mixed with some variation of genetic algorithm to provide initial network weights.

The last and possibly the most exciting area of development is the application of the presented network with the RL algorithm to the real world control scenarios. Since the experiments in this study showed that the network could encode slow sinusoidal waves in pulse density encoding, a similar encoding could be used to moderate the position of robot joints in a similar way to the control of biological muscles. This encoding could be used to control a simple two joint crawling robot. Since the reward of moving forward would be sparse and experiencing distant time lag, the presented network here would be ideally suited to that kind of problem.

Conclusions

The results presented in this thesis suggest that the novel network architecture can predict exponentially growing temporal dependencies with only a linear growth of network depth. Each layer can learn and decode input signal. The temporal activation circuitry can force time compression, while allowing the reuse of nodes. Each layer outputs representation which can be used for further compression. Node weights can be use bidirectionally: decoding the input signal into the output but also mapping output into possible inputs.

The study has evaluated the network mostly using clean and simple input signals, in order to clearly test and demonstrate mechanisms of information encoding. An important step was also to see how the algorithm performs on more complex and noisier data. The capabilities of this network in exploration of reinforcement learning techniques, showed the effectiveness of those RL techniques in solving POMDP toy benchmark problems. After successful performance on simple problems was proven, experiments with more complex simulated and real world problems should be carried out.

The experiments show that the novel network is effective sequence encoding structure that achieves a minimum distance between an input signal and the winning node connection weights. The experiments also show that they do this with only a linear growth in complexity for an exponential growth in encoding power. This network property significantly reduces the cost of encoding and improves the application potential for a range of temporal problems.

Contributions

The work presented in this thesis introduced a new unsupervised sequence encoding algorithm based on multilayer Self-Organizing maps and local short term memory. Although the algorithm borrows behaviours from algorithms previously developed, the work introduces two new crucial elements.

The first one is a wave delay-line, a type of short term memory. This memory is a simplified physical simulation of propagation of wave in a physical medium. Thanks to an ability to modify parameters of the medium within big ranges inside the simulation, the propagation time and therefore the retention time can be easily manipulated. When the retention time of the memory is increased the temporal resolution decreases. This enables the network to deal with temporal noise (jitter) since the timing of the events at higher levels of the network gets partially blurred by the lower temporal resolution of this memory.

The second novel element is a temporal inhibition circuit. The TIC is an additional circuit which orchestrates activation and learning of nodes in time but allowing the nodes to be flexible in their activation depending on the input data. The TIC by managing activation of nodes, spreads resources of long term memory evenly within a temporal sequence. Simultaneously by its flexibility in node activation it solves the problem of temporal misalignment of encoded input sequence.

The last contribution is the development of ways of adopting the presented multi-layer sequence encoder as a model building mechanism to help with hidden state identification in the Reinforcement Learning context. This allows the network to build internal representation of the problem during random exploration. While simultaneously attaching the value to the temporally-extended traces of action-states. A network built in this manner was shown to be able to then be decoded for optimal actions in the temporal context.

Acronyms

BMU	Best matching unit. 35
HSOM	Hierarchical self-organising map. 28
HTM	Hierarchical temporal memory. 23
MDP	Markov decision process. 28
PBVI	Point based value iteration. 18
POMDP	Partially observable Markov decision process. 17
RL	Reinforcement Learning. 17
RNN	Recurrent Neural Network. 22
SOM	Self-organising map. 26
TIC	Temporal inhibition circuit. 41

Bibliography

Aaron van den Oord, e. a. (2016). WaveNet: A Generative Model for Raw Audio. *arXiv Prepr. arXiv:1609.03499v2*.

Bai, S., Kolter, J. Z., & Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling.

Barto, A., & Mahadevan, S. (2002). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems: Theory and Applications*, *13*.

Bellman, R. (1953). *An Introduction to the Theory of Dynamic Programming*. Project Rand R-245. Rand Corporation.
URL `https://books.google.co.uk/books?id=SmElBX4jq6wC`

c. Chappelier, J., & Grumbach, A. (1996). A kohonen map for temporal sequences. In *In Proceedings of NEURAP'95*, (pp. 104–110).

Cassandra, A. R. (1998). A survey of pomdp applications. In *Working notes of AAAI 1998 fall symposium on planning with partially observable Markov decision processes*, vol. 1724.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., nd Holger Schwenk, F. B., & Bengio, Y. (2014). Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*.

Cohen, L., Chaput, H., & Cashon, C. (2002). A constructivist model of infant cognition. *Cognitive Development*, (pp. 1323–1343).

Cui, Y., Ahmad, S., & Hawkins, J. (2017). The htm spatial pooler—a neocortical algorithm for online sparse distributed coding. *Frontiers in Computational Neuroscience*, *11*.
URL `https://www.frontiersin.org/article/10.3389/fncom.2017.00111`

Dai, N., Liang, J., Qiu, X., & Huang, X. (2019). Style transformer: Unpaired text style transfer without disentangled latent representation. In *ACL*.

Deng, L. (2014). A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA Transactions on Signal and Information Processing*, *3*, e2.

Farkas, I., Bosák, R., & Gergel, P. (2016). Computational analysis of memory capacity in echo state networks. *Neural networks : the official journal of the International Neural Network Society*, *83*, 109–120.

Ghalib, H., & Huyck, C. (2007). A cell assembly model of sequential memory. In *2007 International Joint Conference on Neural Networks*, (pp. 625–630).

Heideman, M. T., Johnson, D. H., & Burrus, C. S. (1984). Gauss and the history of the fast Fourier transform. *IEEE ASSP Magazine*, *1*, 14–21.

Hochreiter, S. (1998). Recurrent neural net learning and vanishing gradient. *International Journal Of Uncertainity, Fuzziness and Knowledge-Based Systems*, *6*(2), 107–116.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Humble, J., Denham, S., & Wennekers, T. (2012). Spatio-temporal pattern recognizers using spiking neurons and spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience*, *6*.
URL `https://www.frontiersin.org/article/10.3389/fncom.2012.00084`

Izhikevich, E. M. (2006). Polychronization: Computation with spikes. *Neural computation*, *18*, 245–82.

Jaeger, H. (2001). The "echo state" approach to analysing and training recurrent neural networks-with an erratum note'. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, *148*.

Kamat, A., & Precup, D. (2020). Diversity-enriched option-critic.

Kangas J.A., L. J., Kohonen T.K. (1990). Variants of self-organizing maps. *IEEE Transactions on Neural Networks*, *1*, 93–99.

Kohonen, T. (1982). Self-Organized Formation of Topologically Correct Feature Maps. *Biological Cybernetics*, *43*, 59–69.

Kohonen, T. (1989). *Self-Organization and Associative Memory: 3rd Edition*. Berlin, Heidelberg: Springer-Verlag.

Le, Q. V., Jaitly, N., & Hinton, G. E. (2015). A simple way to initialize recurrent networks of rectified linear units. *CoRR*, *abs/1504.00941*.

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*, 2278 – 2324.

Li, Z., Liu, F., Yang, W., Peng, S., & Zhou, J. (2021). A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, (pp. 1–21).

Maass, W. (1997). Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, *10*, 1659–1671.

Maass, W., Natschläger, T., & Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, *14*, 2531–60.

Maes, A., Barahona, M., & Clopath, C. (2020). Learning spatiotemporal signals using a recurrent spiking network that discretizes time. *PLOS Computational Biology*, *16*, 1–26.
URL `https://doi.org/10.1371/journal.pcbi.1007606`

Martinetz, T., & Schulten, K. (1991). A "neural-gas" network learns topologies. *Artificial neural networks*, *1*, 397–402.

McCallum, R. A. (1996). Hidden state and reinforcement learning with instance-based state identification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *26*(3), 464–473.

Mcqueen, T., Hopgood, A., Tepper, J., & Allen, T. (2003). A recurrent self-organizing map for temporal sequence processing. *Applications and Science in Soft Computing, Ser. Advances in Soft Computing*, *24*, 3–8.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*, 529–533.

Nachum, O., Tang, H., Lu, X., Gu, S. S., Lee, H., & Levine, S. (2019). Why does hierarchy (sometimes) work so well in reinforcement learning? *ArXiv*, *abs/1909.10618*.

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

Pierris, G., & Dahl, T. (2017). Learning robot control using a hierarchical SOM-based encoding. *IEEE Transactions on Cognitive and Developmental Systems*, *9*, 30–43.

Pineau, J., Gordon, G., & Thrun, S. (2006). Anytime point-based approximations for large pomdps. *J. Artif. Int. Res.*, *27*(1), 335–380.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408.

Schmidhuber, J. (2014). Deep learning in neural networks: An overview. *Neural Networks*, *61*.

Schrauwen, B., Verstraeten, D., & Campenhout, J. (2007). An overview of reservoir computing: Theory, applications and implementations. (pp. 471–482).

Sheik, S., Pfeiffer, M., Stefanini, F., & Indiveri, G. (2013). Spatio-temporal spike pattern classification in neuromorphic systems. In N. F. Lepora, A. Mura, H. G. Krapp, P. F. M. J. Verschure, & T. J. Prescott (Eds.) *Biomimetic and Biohybrid Systems*, (pp. 262–273). Berlin, Heidelberg: Springer Berlin Heidelberg.

Thorpe, S., Delorme, A., & van Rullen, R. (2001). Spike-based strategies for rapid processing. *Neural networks : the official journal of the International Neural Network Society*, *14 6-7*, 715–25.

Varstal, M., Millán, J. D. R., & Heikkonen, J. (1997). A recurrent self-organizing map for temporal sequence processing.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. NIPS'17, (p. 6000–6010). Red Hook, NY, USA: Curran Associates Inc.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics*, *37*(3), 328–339.

Wan, E. (1990). Temporal backpropagation for fir neural networks. In *1990 IJCNN International Joint Conference on Neural Networks*, (pp. 575–580 vol.1).

Wang, W., Pedretti, G., Milo, V., Carboni, R., Calderoni, A., Ramaswamy, N., Spinelli, A. S., & Ielmini, D. (2018). Learning of spatiotemporal patterns in a spiking neural network with resistive switching synapses. *Science Advances*, *4*(9), eaat4752.
URL `https://www.science.org/doi/abs/10.1126/sciadv.aat4752`

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, King's College, Oxford.

Williams, R. J., & Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, *2*, 490–501.

Y. Cui, J. H., S. Ahmad (2015). Continuous online sequence learning with an unsupervised neural network model. *arXiv Prepr. arXiv1512.05463*.

Y. Yamashita, J. T. (2008). Emergence of functional hierarchy in a multiple timescale neural network model: a humanoid robot experiment. *PLoS Computational Biology*, *4*(11).

You, J., Wang, Y., Pal, A., Eksombatchai, P., Rosenberg, C., & Leskovec, J. (2019). Hierarchical temporal convolutional networks for dynamic recommender systems.

Zhang, W., Itoh, K., Tanida, J., & Ichioka, Y. (1990). Parallel distributed processing model with local space-invariant interconnections and its optical architecture. *Appl. Opt.*, *29*(32), 4790–4797.
URL `https://opg.optica.org/ao/abstract.cfm?URI=ao-29-32-4790`

Appendices

Appendix A

Short-term wave delay line memory code

Below is the implementation of the code performing short-term wave delay line memory functions:

```
classdef LIN_MEM < handle
   properties
       x;
       xdot;
       dt;
       k;
       d;
   end

   methods
       function obj = LIN_MEM(size,timeHorizonVal,retentionTimeVal)
           obj.x = zeros(size,timeHorizonVal);
           obj.xdot = zeros(size,timeHorizonVal);
           obj.k = 1.0;
           obj.dt = 0.1/retentionTimeVal;
           obj.d = 0.14;
       end

       function step(obj,dim)

           if dim~=0
               obj.x(dim)=1;
```

```
            end

            for i=1:10
                dif = obj.x(:,1:end-1)-obj.x(:,2:end);
                f = [zeros(size(dif,1),1),dif]+[-dif,zeros(size(dif
    ,1),1)];

                obj.xdot(:,:) = obj.xdot(:,:)+(obj.dt*obj.k*f);
                obj.xdot(:,end) = (1-obj.d)*obj.xdot(:,end);
                obj.x(:,:) = obj.x(:,:)+obj.dt*obj.xdot(:,:);
                obj.x(obj.x(:)<0)=0.0;
            end
        end
    end
end
```

Appendix B

Self-organising map code

Below is the implementation of the code performing self-organising map functions:

```
classdef SOM < handle
    properties
         inputSize;
         latticeSize;
         mapWeights;
%         BMU;
         TBMU;
         dW;
    end
    methods
         function obj = SOM(laticeSizeVal,inputSizeVal)
             obj.inputSize = inputSizeVal;
             obj.latticeSize = laticeSizeVal;
             obj.mapWeights= 0.1*rand(obj.latticeSize(1),obj.
   latticeSize(2),obj.inputSize);
             obj.TBMU = zeros(4,1);
             obj.dW = 0;
         end

         function findBMUEucliMis(obj,input)
             weights = reshape(obj.mapWeights,[],obj.inputSize);
             euclideanDistance = ( ones(size(weights,1),1)*input(:)
   ' - weights ).^2;
```

```
          euclideanDistance(isnan(euclideanDistance)) = 0;
          euclideanDistance = sum(euclideanDistance,2);
           %dotProduct = (reshape(obj.mapWeights,[],obj.inputSize)*input(:))./L(:);

            [obj.TBMU(4),obj.TBMU(3)] = min(euclideanDistance(:));
            [obj.TBMU(1),obj.TBMU(2)] = ind2sub(obj.latticeSize,obj.TBMU(3));

      end

      %adjust weights of map according to Gausian neighborhood function
      function adjWeightsN(obj,BMU,input,varience,learningRate)

          %find distance to BMU
         ColInd = 1:obj.latticeSize(1);
         RowInd = 1:obj.latticeSize(2);
         ColInd = (ColInd-BMU(1)).^2;
         RowInd = (RowInd-BMU(2)).^2;
         Distance = ((ColInd'*ones(1,obj.latticeSize(2)))+(ones(obj.latticeSize(1),1)*RowInd));

          %reshape map weghts to 2D
         w = reshape(obj.mapWeights,[],obj.inputSize);
          %apply neighborhood function Gaussian
         l = learningRate*exp((-1*Distance(:))/(2*varience^2));
         %finde the weights change
         dw = ( diag(l) * (  ones(size(w,1),1)*(input(:)' ) - w ) );
          %addjust weights
         w = w + dw;
          %reshape mapWeight back to 3D
         obj.mapWeights = reshape(w,obj.latticeSize(1),obj.latticeSize(2),[]);
         %remmember change in weights
```

```
            obj.dW = sum(sum(abs(dw)));
        end

    end
end
```

Appendix C

Single layer code

Below is the implementation of the code performing single layer network functions, joining short-term memory, self-organising map and temporal inhibition circuit. The "LEYER2" class is design to work with the new wave-delay line memory while the simpler "LEYER" class incorporates simple tape delay line memory:

```
classdef LEYER2 < handle
   properties
        SOMleyer;
        leyerOS;
        inputSize;
        timeHorizon;
        inputMap;
        BMUMap;
        inhib;
        lastBMUMap;
        mem;
        retention;
        treshold;
   end

   methods
        function obj = LEYER2(mapSizeVal,inputSizeVal,
   timeHorizonVal,retentionVal,tresholdVal)
            obj.inputSize = inputSizeVal;
            obj.timeHorizon=timeHorizonVal;
            obj.retention = retentionVal;
%           obj.inputMap = zeros(obj.inputSize,obj.timeHorizon*3);

            obj.SOMleyer = SOM(mapSizeVal,obj.inputSize*obj.
timeHorizon);
%           obj.BMUMap = zeros(4,obj.timeHorizon+1);
            obj.leyerOS = mapSizeVal(1)*mapSizeVal(2);

            obj.BMUMap = zeros(4,1);
            obj.inhib = +inf;
            obj.lastBMUMap = ones(mapSizeVal);

            obj.mem = LIN_MEM(obj.inputSize,timeHorizonVal,obj.
retention);
            obj.treshold = tresholdVal;
        end

        function processInput(obj,inputBMU,n,l)

%             obj.inputMap(:,1:obj.timeHorizon*3-1) = obj.inputMap
(:,2:obj.timeHorizon*3);
%             obj.inputMap(:,obj.timeHorizon*3) = zeros(obj.
inputSize,1);
%             obj.inputMap(:,obj.timeHorizon*2) = zeros(obj.
inputSize,1);
%             if inputBMU ~= 0
%                 obj.inputMap(inputBMU,obj.timeHorizon*2) = 1.0;
%             end

            if inputBMU ~= 0
                obj.mem.step(inputBMU);
            else
                obj.mem.step(0);
            end

            Input = obj.mem.x;
            obj.SOMleyer.findBMUEucliMis(Input);

            %update nenory of TBMU
            obj.BMUMap(:,1) = obj.SOMleyer.TBMU;

            %check whether node should be selected
            if (obj.BMUMap(4,1) > obj.treshold || obj.lastBMUMap(
obj.BMUMap(3,1)) < obj.timeHorizon*obj.retention) && obj.inhib <
 obj.timeHorizon*obj.retention
            %if obj.BMUMap(4,obj.timeHorizon+1) > 0.1 && obj.inhib
+1 < obj.timeHorizon
                obj.BMUMap(:,1) = zeros(4,1); %clean last BMU
                obj.inhib  = obj.inhib + 1;
                obj.lastBMUMap(:) = obj.lastBMUMap(:)+1;
            else
                obj.inhib = 1;
                obj.lastBMUMap(obj.BMUMap(3,1)) = 1;
            end

            if l>0 && n>0
                 if obj.BMUMap(3,1) ~= 0 %check whether there was
BMU activation
                     Input = obj.mem.x;
                     %Input = obj.inputMap(:,1:obj.timeHorizon*2);
                     obj.SOMleyer.adjWeightsN(obj.BMUMap(:,1),Input
,n,l);
                 end
            end


        end
    end
end
```

```
classdef LEYER < handle
    properties
        SOMleyer;
        leyerOS;
        inputSize;
        timeHorizon;
        inputMap;
        BMUMap;
        inhib;
        lastBMUMap;
    end

    methods
        function obj = LEYER(mapSizeVal,inputSizeVal,timeHorizonVal
)
            obj.inputSize = inputSizeVal;
            obj.timeHorizon=timeHorizonVal;

            obj.inputMap = zeros(obj.inputSize,obj.timeHorizon*3);
            obj.SOMleyer = SOM(mapSizeVal,obj.inputSize*obj.
timeHorizon*2);
            obj.BMUMap = zeros(4,obj.timeHorizon+1);
            obj.leyerOS = mapSizeVal(1)*mapSizeVal(2);
            obj.inhib = +inf;
            obj.lastBMUMap = ones(mapSizeVal);
        end

        function processInput(obj,inputBMU,n,l)
            obj.inputMap(:,1:obj.timeHorizon*3-1) = obj.inputMap
(:,2:obj.timeHorizon*3);
            obj.inputMap(:,obj.timeHorizon*3) = zeros(obj.inputSize
,1);
            obj.inputMap(:,obj.timeHorizon*2) = zeros(obj.inputSize
,1);
            if inputBMU ~= 0
                obj.inputMap(inputBMU,obj.timeHorizon*2) = 1.0;
            end

            %find best matching node in som using missing data
            misInput = obj.inputMap(:,obj.timeHorizon+1:obj.
timeHorizon*3);
        misInput(:,obj.timeHorizon+1:obj.timeHorizon*2) = NaN(obj.inputSize,obj.timeHorizon);
        obj.SOMleyer.findBMUEucliMis(misInput);

        %update nenory of TBMU
        obj.BMUMap(:,1:obj.timeHorizon) = obj.BMUMap(:,2:obj.timeHorizon+1);
        obj.BMUMap(:,obj.timeHorizon+1) = obj.SOMleyer.TBMU;


        %check whether node should be selected
        if (obj.BMUMap(4,obj.timeHorizon+1) > 0.5 || obj.lastBMUMap(obj.BMUMap(3,obj.timeHorizon+1)) < obj.timeHorizon) && obj.inhib < obj.timeHorizon
        %if obj.BMUMap(4,obj.timeHorizon+1) > 0.1 && obj.inhib+1 < obj.timeHorizon
            obj.BMUMap(:,obj.timeHorizon+1) = zeros(4,1); % clean last BMU
            obj.inhib  = obj.inhib + 1;
            obj.lastBMUMap(:) = obj.lastBMUMap(:)+1;
        else
            obj.inhib = 1;
            obj.lastBMUMap(obj.BMUMap(3,obj.timeHorizon+1)) = 1;
        end

        %if new node was selected do a input prediction
        if obj.BMUMap(3,obj.timeHorizon+1) ~= 0 %check whether there was BMU activation
            w = obj.SOMleyer.mapWeights(obj.BMUMap(1,obj.timeHorizon+1),obj.BMUMap(2,obj.timeHorizon+1),:);
            w = reshape(w,obj.inputSize,[]);
            obj.inputMap(:,obj.timeHorizon*2+1:end) = w(:,obj.timeHorizon+1:end);
        end

```

```
            %learn full representation for node activated
   timeHorizon ago
            if l>0 && n>0
                if obj.BMUMap(3,1) ~= 0 %check whether there was
   BMU activation
                    Input = obj.inputMap(:,1:obj.timeHorizon*2);
                    obj.SOMleyer.adjWeightsN(obj.BMUMap(:,1),Input,
   n,1);
                end
            end

        end
   end
end
```

Appendix D

Single convergence activation test code

Below is the implementation of the code performing single convergence activation test from experiment 3.3:

```
N = 5000;

F = 1/50;
qe = 0;
g = 1;
code = zeros(1,N);
inputval = zeros(1,N);

output0 = zeros(2,N,3);
output1 = zeros(9,N);
output2 = zeros(9,N);
output3 = zeros(4,N);
output4 = zeros(4,N);
outputN = zeros(4,N);
outputED = -1*ones(4,N);

dwOutput = zeros(4,N);
lOutput = zeros(4,N);

pos = zeros(2,N);

%leyer1;
```

```

MapSize = [3,3];
leyer1 = LEYER(MapSize,2,4);
MapSize = [3,3];
leyer2 = LEYER2(MapSize,leyer1.leyerOS,8,1,0.35);
%
MapSize = [2,2];
leyer3 = LEYER2(MapSize,leyer2.leyerOS,8,2,0.5);


for n=2:N



    %first order delta-sigma modulator
    inputval(n) = g*sin(n*F*2*pi);%+0.001*randn(1);
    if inputval(n)>=qe
        code(n) = 1;
    else
        code(n) = -1;
    end
    qe = code(n)-inputval(n)+qe;

    input = (code(n)>0)+1;
    %input = randi([1 2]);
    %-------------------------leyer1---------------------------

         output0(input,n,1) = 1;

        %first leyer part%
        LearnRate = exp(-5.0*n/(N));


        leyer1.processInput(input,0.8*LearnRate,0.5*LearnRate);
        dwOutput(1,n) = leyer1.SOMleyer.dW;
        lOutput(1,n) = 0.5*LearnRate;
        if leyer1.BMUMap(3,leyer1.timeHorizon+1) ~= 0
```

```
            output1(leyer1.BMUMap(3,leyer1.timeHorizon+1),n) = 1;
            outputN(1,n) = leyer1.BMUMap(3,leyer1.timeHorizon+1);
            outputED(1,n) = leyer1.BMUMap(4,leyer1.timeHorizon+1);
            %conn = leyer1.BMUMap(3,leyer1.timeHorizon+1);
        end

        %second leyer
        LearnRate = exp(-4.0*(n)/(N));


        leyer2.processInput(leyer1.BMUMap(3,leyer1.timeHorizon+1)
    ,0.8*LearnRate,0.5*LearnRate);
        dwOutput(2,n) = leyer2.SOMleyer.dW;
        lOutput(2,n) = 0.5*LearnRate;
        if leyer2.BMUMap(3,1) ~= 0
            output2(leyer2.BMUMap(3,1),n) = 1;
            outputN(2,n) = leyer2.BMUMap(3,1);
            outputED(2,n) = leyer2.BMUMap(4,1);
        end

         %third leyer
         LearnRate = exp(-3.0*(n)/(N));

        leyer3.processInput(leyer2.BMUMap(3,1),0.8*LearnRate,0.5*
    LearnRate);
        dwOutput(3,n) = leyer3.SOMleyer.dW;
        lOutput(3,n) = 0.5*LearnRate;
        if leyer3.BMUMap(3,1) ~= 0
            output3(leyer3.BMUMap(3,1),n) = 1;
            outputN(3,n) = leyer3.BMUMap(3,1);
            outputED(3,n) = leyer3.BMUMap(4,1);
        end


    if mod(100*n/N,1)==0
         fprintf('progress = %0.0f%%\n',100*n/N);
    end
```

```

end

close all
plotSpikes

figure;
plotWeights(leyer1);
figure;
plotWeights(leyer2);
figure;
plotWeights(leyer3);
```

Appendix E

Encoding for classification code

Below is the implementation of the code performing training and testing for classification from experiment 3.5:

```
testMod = false;

if testMod == 1
    NN = 10000;
    load('test.mat');
    test(:,785) = test(:,785)+1;
else
    NN = 60000;
    load('train.mat');
    train(:,785) = train(:,785)+1;
end

N=784*NN;


if testMod == 0
    readSeq = randperm(60000);
else
    readSeq = randperm(10000);
end

insize = 392;
```

```

%initialize layers
if testMod==0
    layer1 = SOM([1,2],1);
    layer2 = LEYER([3,3],2,4);
    layer3 = LEYER2([3,3],layer2.leyerOS,8,1,0.35);
    layerEnd = SOM([15,15],layer3.leyerOS*insize+10);
end

mem = LIN_MEMF(layer3.leyerOS,insize,784/insize);

output = zeros(NN,2);
output1 = zeros(N,2);
output2 = zeros(N,layer2.leyerOS);
output3 = zeros(N,layer3.leyerOS);

InputScale = 20;

for n = 1:N

    %select new digit for trining
    if mod(n,784)==1
        digit = readSeq(floor(n/784)+1);
    end

    %first layer quantization of pixel intensity
    if testMod==1
        layer1.findBMUEucliMisFull(test(digit,mod(n-1,784)+1));
    else
        LearnRate = exp(-3*n/(N));
        layer1.findBMUEucliMisFull(train(digit,mod(n-1,784)+1));
        layer1.adjWeightsN(layer1.TBMU(1:2),train(digit,mod(n
   -1,784)+1),1.0*LearnRate,.1*LearnRate);
    end

    output1(n,layer1.TBMU(3)) = 1;

```

```matlab
    %second layer
    LearnRate = exp(-2*n/(N));
    if testMod==1
        LearnRate = 0;
    end
    layer2.processInput(output1(n,:),0.5*LearnRate,.1*LearnRate);

    if layer2.BMUMap(3,layer2.timeHorizon+1) ~= 0
             output2(n,layer2.BMUMap(3,layer2.timeHorizon+1)) = 1;
    end

    %third layer
    LearnRate = exp(-1*n/(N));
    if testMod==1
        LearnRate = 0;
    end
    layer3.processInput(layer2.BMUMap(3,layer2.timeHorizon+1),0.5*LearnRate,.1*LearnRate);

    if layer3.BMUMap(3,1) ~= 0
            output3(n,layer3.BMUMap(3,1)) = 1;
    end

    %add third layer autput to clasification SOM memory
    mem.step(layer3.BMUMap(3,1));

    %run clasification on the end of the sequence
    if mod(n-1,784)+1 == 784

        %prediction
        ED = layerEnd.findBMUEucliMisFull([mem.x(:);NaN(10,1)]);

        %consider only nodes with strongly defined class (>.75)
        [v,~] = max(layerEnd.mapWeights(:,:,end-9:end),[],3);
        v = 1*(v>0.75*InputScale);
        v(v==0) = +inf;
        ED = v(:).*ED;
```

```

     %find class of clossest node
     [~,layerEnd.TBMU(3)] = min(ED);
     [layerEnd.TBMU(1),layerEnd.TBMU(2)] = ind2sub(layerEnd.
latticeSize,layerEnd.TBMU(3));
     [~,predDig] = max(layerEnd.mapWeights(layerEnd.TBMU(1),
layerEnd.TBMU(2),end-9:end));

     %save predicted and real pattern class
     if testMod == 1
         output(floor(n/784),:) = [predDig,test(digit,785)];
     else
         output(floor(n/784),:) = [predDig,train(digit,785)];
     end

     %learning
     if testMod == 0
         input = zeros(10,1);
         input(train(digit,785)) = InputScale;

         LearnRate = exp(-1*n/(N));
         layerEnd.findBMUEucliMisFull([mem.x(:);input]);
         layerEnd.adjWeightsN(layerEnd.TBMU(1:2),[mem.x(:);input
],1.2,.1*LearnRate);
     end

     mem.clean;
 end



 if mod(100*n/N,1)==0
     fprintf('progress = %0.0f%%\n',100*n/N);

     for i =1:10
         subplot(1,10,i);
         drawMap(layerEnd.mapWeights(:,:,end-10+i)/(.75*
InputScale));
```

```
        end

    end

end

%plot prediction stats
nbins = 100;
stats = zeros(1,nbins);
for i=1:nbins
 bin = NN/nbins;
 stats(i) = sum(output((i-1)*bin+1:i*bin,1)==output((i-1)*bin+1:i*
   bin,2))/bin;
end
figure;
plot(stats);
ylim([0 1]);

%plot map classes in color
[v,cmap] = max(layerEnd.mapWeights(:,:,end-9:end),[],3);
colors = [(0.5+0.5*sin((.15*pi*(0:9))+(2/3*(2*pi))));
    (0.5+0.5*sin((.15*pi*(0:9))+(1/3*(2*pi))));
    (0.5+0.5*sin((.15*pi*(0:9))+(0/3*(2*pi))))];
gray = zeros(29*layerEnd.latticeSize(1)+1,29*layerEnd.latticeSize
   (1)+1,3);
for i=1:layerEnd.latticeSize(1)
    for j=1: layerEnd.latticeSize(2)
        %d = reshape(layerEnd.mapWeights(i,j,1:784),28,[])';
        %d = reshape(d,[],1);
        d = ones(784,1);
        d = d*(v(i,j)>.75*InputScale);
        d = squeeze(d)*colors(:,cmap(i,j))';
        d = reshape(d,28,28,[]);
        gray(29*(i-1)+2:29*i,29*(j-1)+2:29*j,:) = d;
        %gray(29*(i-1)+2:29*i,29*(j-1)+2:29*j) = reshape(layerEnd.
   mapWeights(i,j,:),28,[])';
```

```
        %subplot( layerEnd.latticeSize(1),layerEnd.latticeSize(1),(
    i-1)* layerEnd.latticeSize(1)+j)
        %drawMap(
    end
end
figure;
imshow(gray,'InitialMagnification','fit');
```