MASTER THESIS

Sequence processing and learning with a reservoir computing approach

By Lara Janßen s1838105

Faculty of Behavioural, Management and Social sciences

EXAMINATION COMMITTEE

Frank van der Velde
Willem Verwey

DOCUMENT NUMBER

BMS - NUMBER

23/12/2020

Abstract

Sequence learning is an inherent human ability, which plays a critical part in human intelligence. The brain is capable of rapidly processing and learning sequential information. Since so many aspects of human cognition deal with sequence learning, the question was raised how neural sequence learning and processing can be simulated. Research suggests that the use of chunks, defined as a memory representation of a basic sequence of items, can make sequential learning more efficient. This shows that the learning of chunks plays a vital role in sequential behaviour and learning. The focus of this study is to simulate how these basic sequences can be learned with a neural network, based on the assumption that such a sequence is just an association between items. In particular, this project aims to understand how such a basic sequence can be learned in a sequential manner, which means presenting each item of a sequence one by one and learn during that process. With the use of reservoir computing a network was built. In the standard approach of reservoir computing, the sequential nature of the network lies within the reservoir, given by random, sparse, and fixed connections between the nodes in the reservoir. In this way, the reservoir offers a set of fixed and randomly organized sequences, which can be used for sequence learning. Learning then occurs in the connections between the reservoir and the output nodes. However, the initial simulation results casted doubts on the cognitive ability of the reservoir computing approach to address human sequential learning because it failed to learn a basic sequence in a sequential manner. To solve this issue a new approach is introduced, in which the role of learning in reservoir computing is changed. By learning sequences within the reservoir instead of from the reservoir to the output nodes a basic sequence could be learned in a sequential manner. The network simulated the learning and reactivation of five sequences with twenty items within the reservoir. In this way, a first step was made towards a better approach for modeling human sequential learning.

Keywords: *Sequential learning; Reservoir computing; Neural networks; Chunking*

Table of Contents

1.0 Introduction

Sequence learning is an inherent human ability and a critical part of human intelligence. One defining feature of the brain is the ability to produce and recognize ordered sequences, which is part of many cognitive performances, from sequencing sound to speech, playing an instrument or performing motor tasks, like driving a car (Clegg, DiGirolamo, & Keele, 1998). Already early in an infant's life, it is able to learn the sequential structure of syllable sequences and he or she is capable of detecting new sequences of the same syllables even when these sequences violate previously learned sequences (Saffran, Aslin, & Newport, 1996).

Many processes of sequence learning and production are hierarchical processes, which can be observed in speech organization, behavioural sequences and thought processes (Fonollosa, Neftci, & Rabinovich, 2015). One example for such hierarchical processes can be found in human language, specifically when looking at syntax (Conway, 2012). Given a sentence "The dog catches the ball", it is necessary not just to recognize the order of the words but also their relationship. The brain creates hierarchical representations even when presented with sequentially presented input (Uddén, de Jesus Dias Martins, Zuidema, & Tecumseh Fitch, 2019). This has been demonstrated in several theories of language processing and speech organization (Ellis & Sinclair, 1996; Gee & Grosjean, 1983).

One mechanism that makes it easier to retain and recall information is to segment a sequence of elements into blocks or chunks (Ericcson, Chase, & Faloon, 1980). Miller (1956) originally introduced the idea of a memory chunk which he defined as "a memory symbol with which several memory items can be treated as a single processing unit". Miller (1956) pointed out the limitations of human working memory capacity for information processing, which necessitated the organization of elements into chunks.

Another common definition describes a memory chunk as a collection of elements with strong associations with one another but weak associations with elements in other chunks (Gobet et al., 2001). One example is complex motor movements, which are represented as a chain of memory representations or motor chunks, these are linked in a goal-specific way (Verwey, 2001). Further, behavioural visuo-motor sequence learning experiments showed that motor sequences are organized as chunks each of which are represented by a single memory unit (Bo & Seidler, 2009; Pammi et al., 2012; Sakai, Kitaguchi, & Hikosaka, 2003; Wymbs, Bassett, Mucha, Porter, & Grafton, 2012).

Chunking has been shown in a wide range of instances. Imaging and behavioural studies show that chunking learning extends to language processing, visual perception, habit learning and motor skill acquisition (Ellis & Sinclair, 1996; Gee & Grosjean, 1983; Graybiel, 1998; Luck & Vogel, 1997; Rosenbaum, Kenny, & Derr, 1983).

Furthermore, research into motor behaviour shows that people form “motor chunks”. This can be explained by the dual processor model, which states that representations are used by two independent, parallel processors (Verwey, 2001). The motor processor which executes more tightly coupled elements of a motor chunk and the cognitive processor which links these chunks together into longer sequences (Verwey, 2001). Another way the cognitive processor deals with motor chunks is to increase execution rate of movements in parallel to the motor processor by drawing on explicit knowledge (Verwey & Eikelboom, 2003). This happens either because the sequence is short or when the last chunk is already initiated (Verwey, 2015)

Studies on chunking in motor behaviour show that motor chunks are robust and stable over time. Verwey (2001) found that this phenomenon occurred in different situations in two experimental studies in which participants practiced a series of key press sequences. This shows that chunking is a part of motor behaviour and plays a role in learning. Another study by Sakai et al. (2003) investigated the temporal pattern in which participants performed visuomotor sequences. They found that participants learned the visuomotor sequences by spontaneously representing short movement series into single motor chunks. The chunking pattern varied between individuals, even when the same motor sequence was performed (Sakai et al., 2003). Further, they showed that chunking makes visuomotor sequence more efficient. The spontaneous nature of the memory chunk formation is similar to the emergence of habits. Habits are formed through continuous repetition of the same behaviour and these stay stable over time.

Various brain regions are involved during human sequence learning and chunking behaviour (Verwey, 2019)One brain region associated with the concatenation of motor elements into sequences is the basal ganglia (Rabinovich, Varona, Tristan, & Afraimovich, 2014). The basal ganglia has been shown to facilitate motor sequence learning and habit formation by sequencing motor acts into chunks, which contributes to motor skill acquisition (Salmon & Butters, 1995). This has been supported by studies in patients with Parkinson’s disease and stroke patients. Damage to the basal ganglia impaired one's ability to form accurate motor chunks (Rabinovich et al., 2014).

The cerebellum is also associated with motor skill acquisition and sequence learning. Studies in patients with damage to this brain area indicate that the cerebellum may be involved in the timing and indexing of events which allow the motor movements to be ordered into sequences (Salmon & Butters, 1995). Involvement of different regions involved in the process of sequential learning and chunking behaviour makes it a dynamical system, as different parts work together and are interconnected. With regard to language processing the cortical-striatal system is involved, which is made up of the cortex and striatum and the cortical input nucleus of the basal ganglia (Hinaut & Dominey, 2013). Further studies in patients with damage to the basal ganglia suggest that it is necessary for aspects of syntactic processing.

Another brain region crucial to several forms of learning and memory, but especially for sequence learning, including spatial navigation is the entorhinal-hippocampal circuit (Mehta, 2015). Research in this area showed that sequence learning relies on synaptic plasticity, in particular Hebbian synaptic plasticity. Synaptic plasticity describes changes in synaptic strength over time which is the result of synaptic activity (Ho, Lee, & Martin, 2011). For the expression of synaptic plasticity, both pre-synaptic and post-synaptic mechanisms are contributing. This serves as the basis for the Hebbian learning algorithm, which can be implemented in Artificial neural networks and is discussed in more detail in Section 2.2.

1.1 Aim of the study

The research on sequential learning discussed above shows that the brain is capable of rapidly processing sequential information. Research also suggests that the use of memory or motor chunks, defined as a representation of a basic sequence of items, can make sequential learning more efficient. This shows that the formation of memory chunks plays a vital role in sequential behaviour and learning. Referring back to the definition of memory chunks from Gobet et al. (2001), it seems that memory chunks are sequences of strongly associated elements without any further hierarchical structure. The focus of the present study is to simulate how these basic sequences can be learned with a neural network, based on the assumption that such a sequence is just an association between its items.Recent cognitive models also assume that sequences are made of item to item associations (Logan, 2020 ;Lindsey, 2019) In particular, this project aims to understand how a basic sequence can be learned in a sequential manner, which means presenting each item of a sequence one by one and learn during that process.

The approach used to simulate basic sequences is Reservoir Computing (RC), which will be discussed in detail in the following section. In short, reservoir computing is a machine learning technique that can be used to mimic neural microcircuits in the biological brain using an untrained reservoir of neurons (or nodes) and a trained readout function (Klibisz & MacLennan, 2016). Hinaut and Dominey (2013) used the reservoir computing approach to investigate if the system can be used to learn grammatical structures in real-time. They argued that due to the structure of the reservoir, reservoir computing is a suitable technique to simulate brain processes. Their network was capable to give insight into the analogous real-time processing during human sentence processing as revealed by ERP studies (Hinaut & Dominey, 2013). Features of the reservoir computing approach, such as its random like connection structure, make it attractive as a model for basic aspects of neural processing, as also argued by Hinaut and Dominey (2013). With this approach a model of basic neural circuits underlying sequential behaviour can be built.

The aim of the current study is to investigate whether reservoir computing can be used to simulate sequential learning of basic sequences of items as found in a memory chunk, seen as an association between its items. To answer the question if reservoir computing is a suitable method to simulate basic sequences in a sequential manner a few sub questions need to be answered:

- How can basic sequences be represented in a reservoir computing network?
- Can more sequences be stored in a single reservoir computing (RC) network?
- How does the internal structure of a RC network affect the number of sequences that can be stored in a single RC network?
- How can the sequences be separated? So, how can confusion between sequences in recall be prevented?

In the following section, the basics of neural networks are discussed with a focus on reservoir computing. This is followed by the description of the different networks built to simulate basic sequences, these are described in section 3-7. Section 3 starts with a basic reservoir network, each network version described after is an extension to the previous network. Five network versions were built, and their performance is tested. The last network version simulates learning and reactivation of multiple sequences in the reservoir and concludes with a discussion of the results.

2.0 Artificial neural networks

Machine learning, at its core, uses computer algorithms to process large quantities of data. Different machine learning techniques exist, ranging from simple model-fitting algorithms to large neural networks. The focus of this thesis is on a subset of neural networks techniques known as "reservoir computing". "Reservoir computing defines a class of artificial neural network (ANN) models that mimic neural microcircuits in the biological brain using an un-trained reservoir of neurons and a trained readout function" (Klinibsz and MacLennan, 2016, p1).

Neural networks are loosely based on the structures in a biological brain, the networks consist of neurons, also referred to as nodes, that are connected to other nodes like synapses in the brain (Lukoševičius & Jaeger, 2009). An ANN takes data from an input layer, applies a transformation of this data in a hidden layer, and then return values on an output layer (Klibisz & MacLennan, 2016). This is shown in Figure 1, which shows how the network is trained to learn f (x) = y, where x is the input data and y is a ground-truth property of that data (Klibisz & MacLennan, 2016). The connections in the Neural Network are weighted, therefore the connections between the nodes can vary in strength. Certain nodes are connected more strongly to each other than others, this depends on the weight of the connection, which can have either a positive or a negative value. The inputs (x1, x2, etc.) are connected to the neurons with weights (w1, w2, etc.). The nodes operate by summing up the weight of the input signals. Then the activation function is applied to the net input. Through this, the output can be manipulated using different input weights.

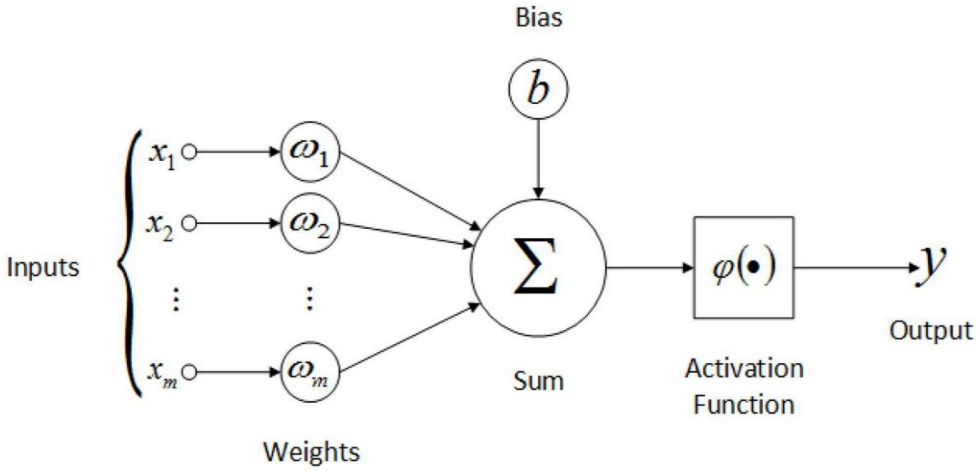

Figure 1 Basic structure of Artificial Neural Network Reprinted from (Ahire, 2018)

By adjusting the computational behaviour of the Artificial Neural Network, it can be trained to learn meaningful data representations through supervised learning where the output matches explicit input labels or unsupervised learning, where the network learns an abstract representation without explicit feedback (Klibisz & MacLennan, 2016).

There are different kinds of Neural Networks, such as feed-forward or recurrent neural networks. They differ in their applications and the kind of connections they allow, but they have three things in common: the individual neuron, the connections between the neurons, and a training algorithm (Vankayala & Rao, 1993).

2.1 Reservoir computing

Reservoir computing (RC) belongs to the recurrent neural networks, the difference to feed-forward networks comes from the way the neurons are connected. In recurrent neural networks the neurons are connected in cycles, rather than layers as seen in feed-forward networks (Lukoševičius & Jaeger, 2009). These cycles make the recurrent neural network a dynamical system. A dynamical system is represented here by a mathematical function that describes how a point in space behaves in time (Sagar, 2019). The connections in the reservoir are random, though fixed and recurrent, corresponding roughly to the structure of the hidden layer in other types of neural networks (Klibisz & MacLennan, 2016). The dynamical nature of recurrent neural networks makes them excel in dealing with time-series data, which can be used for a diverse group of tasks from predicting the weather to controlling chaotic systems (Gauthier, 2018). The difficulty with recurrent neural networks lies in the training, as they are more difficult to train using conventional algorithms for training neural networks (Lukoševičius & Jaeger, 2009).

The neural network approach known as reservoir computing was first proposed by Jaeger (2001) as a simpler way to train Recurrent Neural Networks. He described the Echo State Network (ESN) as “a constructive learning algorithm for recurrent neural networks, which modifies only the weights to output units in order to achieve the learning task” (Jaeger, 2001). Hence, in the standard approach of reservoir computing, the sequential nature of the network lies within the reservoir, given by random, sparse, and fixed connections between the nodes in the reservoir. In this way, the reservoir offers a set of fixed and randomly organized sequences, which can be used for sequence learning. Learning then occurs in the connections between the

reservoir and the output nodes. This approach is very successful for learning sequences (Jaeger, 2001).

When designing a neural network, a number of parameters need to be decided upon. These parameters determine how the neural network is made up and also how it behaves to the inputs. The parameters include for example, the number of layers, the number of neurons per layer and how many training iterations the network goes through (Alto, 2019). RC uses a recurrent neural network and instead of updating all parameters of the network, it only updates part of the parameters, while keeping the others fixed after choosing them randomly (Sagar, 2019). This approach has the advantage that its training is easier and requires less computational power than comparable methods, which make them faster. In the reservoir computing approach, the internal weights are fixed and random, only the output connections (the connections between the reservoir and the output) are changed during training.

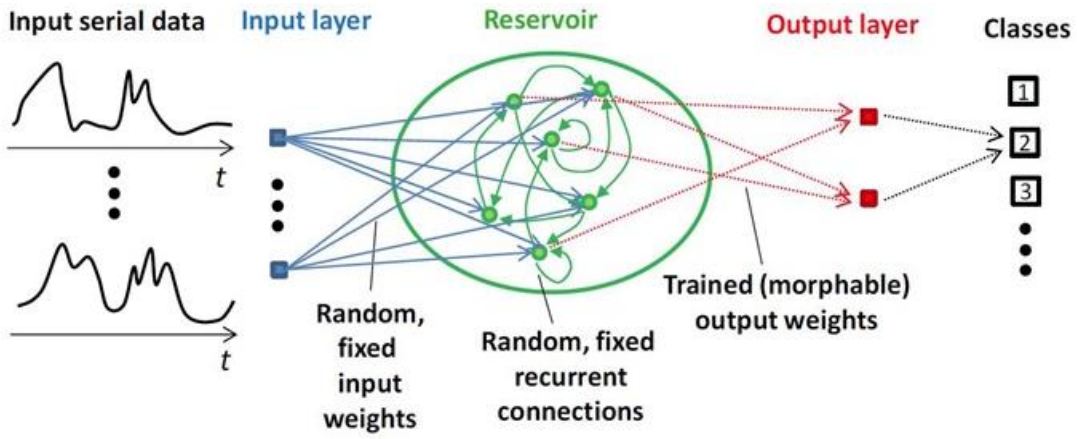

Figure 2 Illustration of Reservoir Computing architecture. Reprinted from Gauthier (2018).

An example of a basic reservoir architecture is shown in Figure 2. The reservoir is composed of a set of nodes that are randomly and sparsely interconnected and that connect to output nodes. The input layer is connected to the reservoir via random, but fixed input weighs. The connections in the reservoir are recurrent and fixed, but random. They store information by connecting the nodes in the reservoir in recurrent loops, where previous inputs affect the next response (Melandri, 2014). The reservoir nodes are connected to the output nodes, this is where learning occurs. The output weights are modifiable and trained. Training methods for the reservoir computing approach are often successful in their task and operate quicker, as not all

layers of the network are trained (Jaeger, 2001). An advantage of the reservoir computing approach compared to other ANN is its capability to process temporal and sequential information with little computational efforts (Hinaut & Dominey, 2013; Klibisz & MacLennan, 2016; Tanaka et al., 2019).

One of the main characteristics of the reservoir computing approach are its recurrent connections, which generate history-dependent dynamical responses to the external inputs (Tanaka et al., 2019). One of the reasons the reservoir is left untrained is that it is based on assumptions about the brain. Cortical microcircuits in the brain are not re-trained or adjusted for different task; they maintain their configuration but re-arrange for different task in other brain areas (Klibisz & MacLennan, 2016). Support for this assumption comes from animal studies. Electrophysiology studies in primates showed that the prefrontal cortex operates on reservoir-like properties (Dominey, 2013). In short, using random connections weights in a structured network reflects the properties of the cortex, with a high predominance of local recurrent connections (Dominey, 2013). These properties seem to make the reservoir computing approach suitable to study and model cognitive processes

The RC model consists of two main components, the reservoir, and a readout function or activation function. The role of the reservoir is to transform the sequential inputs nonlinearly into high-dimensional space in order that features of the inputs can be read out by the learning algorithm (Sagar, 2019). Nonlinearity here describes the response of each element to the input, this is what allows reservoir computers to solve complex problems (Melandri, 2014). The sequential nature of reservoir computing lies within the reservoir through the random connections between the nodes. The connections within the reservoir are intentionally left untrained and the readout function is then used to make sense of the outputs, which means only the last layer of the network is trained, often with a simple learning algorithm such as linear regression (Klibisz & MacLennan, 2016; Tanaka et al., 2019). This mechanism makes the network more computationally efficient and allows to use the item sequences created in the reservoir simultaneously when presented to a learning algorithm.

In the literature on RC there seems to be no universal method for choosing a suitable readout function for the network. Successfully implemented readout functions include single-layer readouts, such as linear regression or support vector machines; multi-layer readouts, such as multi-layer perceptron (Lukoševičius & Jaeger, 2009). Research areas where RC was

implemented include speech and handwriting recognition, robot motor control, time-series prediction, and medical brain-computer interfacing (Klibisz & MacLennan, 2016).

2.2 Hebbian learning rule

The learning rule used for the network in this project is the Hebbian learning rule. Hebbian learning is often described as "neurons that fire together wire together" (Widrow, Kim, Park, & Perin, 2019). This is in an oversimplification of the process; it means that the synaptic weight is increased when two neurons are activated and so the strength of the synapse selectively increases (Widrow et al., 2019). Hebb's postulate is based on a neuro-biological context, as it is based on similar processes that can be observed in the brain, these can be observed in the Hippocampus (Kelso, Ganong, & Brown, 1986; Mehta, 2015). It is similar to the physical changes happening in the brain when something new is learned.

When implemented into a neural network, if two neurons on either side of the synapse are activated simultaneously then the strength of that synapse increases. Reversely, if two neurons on either side of the synapse are activated unsynchronized, the strength of the synapse decreases. Hebbian learning is a form of unsupervised learning (Wallisch et al., 2009). Mathematically it is described as follows:

$$\Delta W_{kj} = L\, x_j\, y_k \qquad [1]$$

In Equation 1 ΔWk_j indicates the change in the synaptic weights between the presynaptic neuron k and the postsynaptic neuron j, L is the learning constant which determines the learning rate, x_j is the activity of the presynaptic neuron, y_k is the activity of the postsynaptic neuron. The Hebbian network is time dependent, which means the weight of the synapse depends on the exact time of the pre and post synaptic activities. It is also a local mechanism, the weight between two neurons A and B depends only on the activities of these two neurons, no other neuron in the network will influence the activation between A and B. This makes the activation correlational; the modification of the synapse is based on the co-occurrence of the pre and post synaptic connections.

3.0 Experiment 1: Initial simulations of sequence learning

The first simulation in this project uses a standard version of the reservoir network, for simulating sequence memorization. The focus is on sequential learning of basic sequences, in which a sequence itself is an input pattern consisting of successively presented individual items. Reservoir computing seems suitable for this task, as sequences are already given in the reservoir. When an input item (given by the activation of a specific input node) is presented to the reservoir it will activate the specific reservoir nodes to which it is connected. This begins the activation of sequence(s) in the reservoir to which these reservoir nodes belong. When more input items are given, they continue and further select a sequence in the reservoir. Hence, one (or more) sequence representations in the reservoir are used to represent a sequence of items presented as input. Initially, the output nodes respond in a random manner to the activation in the reservoir, given by the random weights of the connections between the nodes in the reservoir and the output nodes. By using supervised learning, specific output nodes are selected to represent the items of the sequence. The connections between the reservoirs and the output nodes are then trained to activate the output nodes in the order as represented in the sequence of input items presented to the reservoir. The whole network has learned a sequence correctly when the presentation of the first item of the sequence as input to the reservoir results in the sequential activation of the output nodes that represent the items in the sequence in the correct order.

The aim of this simulation is to sequentially learn a basic sequence consisting of five items. The input items are represented by letters (A, B, C, D, E). Each letter represents an input item of the basic sequence. Hence, there are five input nodes, one for each letter (item), and activated in the order A, B, C, D, E. They activate the nodes in the reservoir in this order as well. In turn, the activated reservoir nodes also activate other reservoir nodes, belonging to the sequences that are stored in the reservoir. There are also five output nodes, one for each of the items A, B, C, D, E. In the training stage, the connections between the reservoir nodes and the output nodes are modified so that the reservoir activates the output node that matches the input presented to the reservoir. For example, if input node C is activated, it activates nodes in the reservoir. The connections of all active nodes in the reservoir (those activated by the input C and those activated by other reservoir nodes, based on the sequences stored in the reservoir) are then trained to activate the output node C. The sequence A, B, C, D, E is learned correctly when the activation of the first input node A results in the activation of all output nodes in the order A, B,

C, D, E. This sequential activation is then produced by the sequential activation of nodes in the reservoir, given by the sequences stored in the reservoir, and initiated by the input node A. Hence, the whole network has memorized the input sequence presented to it.

3.1 Method

Network implementation

The network was built using the programming language Python version 3.6 using the NumPy library. The relevant code can be found in Appendix A.

The network consists of three layers, the input layer, the reservoir, and the output layer. In the input layer no computations are performed as they only pass information to the reservoir (Fumo, 2017). The output layer is connected to the input via the reservoir. It uses the standard way of reservoir learning, the connections in the reservoir are recurrent and fixed and the connections between the reservoir and output are learned.

The network was made by creating connection matrices between layers and within the reservoir. The connections in the reservoir are sparse and randomized. Sparsity gives the chance of having non-zero connections and determines how many connections are formed from the input to the reservoir nodes and from the reservoir nodes to the output. A high sparsity results in fewer connections because the threshold for making a connection is higher. The size of the reservoir can be varied. Connections in the reservoir are possible for each node to every other node except to itself. The model architecture is used to simulate the presentation and learning of a set of sequences in the reservoir (R). The size of the reservoir can be adjusted. Generally, increasing the size of the reservoir increases the number of connections that can be made with the reservoir nodes. A bigger reservoir allows for more connections between the nodes in the reservoir. Hence, it can result in more sequences stored in the reservoir.

Each layer of the model is modelled as an array, except for the input layer. The size of the array is given by the sequence length and size of R. An example of such an array for the output nodes is shown in Table 1. The array consists of two dimensions. One represents the items in the output, which are shown in the columns. The other dimension stands for time, shown in the rows. The time parameter simulates the development of the output nodes over time. It shows the new activation (in the next time step) of the output nodes, based on inputs they receive at a given time

step. The reservoir array is generated in the same manner, each letter belongs to a certain location on the array, but then with the R nodes as the first dimension.

Table 1

Empty array

		Output nodes or reservoir nodes:				
		A	**B**	**C**	**D**	**E**
Time step^0	⟶	0.00	0.00	0.00	0.00	0.00
.		0.00	0.00	0.00	0.00	0.00
.		0.00	0.00	0.00	0.00	0.00
.		0.00	0.00	0.00	0.00	0.00
.		0.00	0.00	0.00	0.00	0.00
.		0.00	0.00	0.00	0.00	0.00
.		0.00	0.00	0.00	0.00	0.00
.		0.00	0.00	0.00	0.00	0.00
.		0.00	0.00	0.00	0.00	0.00
Time step^n	⟶	0.00	0.00	0.00	0.00	0.00

The activation values of the reservoir and the output nodes are given by an adjusted logistic activation function that operates on the sum of the incoming activations. The activation function of a node defines the output of that node, based on the input or a set of inputs (Fumo, 2017). If the input is < 0 or equal to 0 this will result in zero activation, while when input > 0 it results in activation with a maximum value of 1.

Lastly the Hebbian learning rule modifies the output weights of the network. This is necessary for a given input in the network to produce the expected outcome (Fumo, 2017). The output layer is connected to the input via the reservoir. Learning occurs in the connections between the reservoir and output.

Experiment 1

The first experiment (simulation) simulates the evolution of the activation and learning of the output nodes over time. The aim is to learn one sequence consisting of 5 items (A, B, C, D, E). The network consists of 5 input nodes, 10 reservoir nodes and 5 output nodes. Sparsity was set to 0.0. To show the evolution of the activation, 10-time steps were applied. The time steps control how many times the input is presented to the reservoir. The connections from input to output nodes run via one (or more) reservoir nodes.

Item (A) is given first, then item (B) and so forth. So, first item (A) is learned as input and then item (A) is given again to see if the item was learned. The input item (A) activates the reservoir nodes to which it is connected, which then initially activate all output nodes in a random manner. Then, the connections between the reservoir and output node (A) are learned, using Hebbian learning. The next step is to give just item (B) as input and learn the item. Since item (A) was learned in the previous step the sequence should now consist of items (A) and (B). To check if the sequence was learned only item (A) is represented again, which should reactivate the sequence (A, B) as output, through the learned connections in the reservoir. The reservoir nodes activated by item (A) would also activate other R nodes, some of which are connected to the nodes associated with item (B). In this way, a sequence stored in the network is re-activated by presenting its first item to the network. If it was learned as desired the reservoir would have formed connections from the nodes with all nodes in the sequence. This process will be repeated for all other input items until the entire sequence (A, B, C, D, E) is learned.

3.2 Results

The tables below show the evolution of the activation of the output nodes over time for the items (A, B). Each input item in a sequence can be presented to the network and learned separately. Simulations are presented here only for a sequence with two items (A, B), because the simulations showed that it was not possible to sequentially learn a sequence with the current approach.

Table 2 shows the evolution of the activation of output nodes over time for all output nodes. This is given by the development of the output nodes per given time step (shown on the rows). The columns show the activation for the output nodes when the input item (A) is given again after learning the output (A). For the first two time steps the output nodes show no

activation. After three time steps the output node representing item (A) has a higher activation which increases with each time step. After each time step the activation becomes stronger. All other output nodes have no activation as only item (A) was learned for this simulation.

Table 2

Evolution of the activation of the output nodes over time when input A is learned and reactivated

	Output nodes:				
	A	**B**	**C**	**D**	**E**
Time step^0 ⟶	0.00	0.00	0.00	0.00	0.00
.	0.00	0.00	0.00	0.00	0.00
.	0.77	0.00	0.00	0.00	0.00
.	0.95	0.00	0.01	0.00	0.01
.	0.98	0.01	0.01	0.00	0.02
.	0.99	0.01	0.01	0.00	0.02
.	0.99	0.01	0.01	0.00	0.02
.	0.99	0.01	0.01	0.00	0.02
.	0.99	0.01	0.01	0.00	0.02
Time step^n ⟶	0.99	0.01	0.01	0.00	0.02

Note: time is shown in rows; the output nodes are shown on the column

If item (A) was learned, it will be reactivated when item (A) is given again. The input item (A) activates the reservoir nodes associated with this item, which in turn activate the output nodes through their learned connections. This can be visualized by looking at the connections from the reservoir nodes to the output nodes after learning. Table 3 shows the output connections after learning. The rows show the connections from the reservoir to the output nodes (columns). The connections to output node representing item (A) are high. A sequence has been learned if the entire sequence is produced when given the first item of the sequence. Only one item was learned for this first simulation and it was reactivated after learning. The other nodes have no or low connections, as only one input item (A) was presented to the network.

Table 3

Output connections after learning when input node A is learned

	Output nodes:				
	A	**B**	**C**	**D**	**E**
Connections of R	0.90	0.00	0.00	0.00	0.00
Nodes ⟶	1.07	0.00	0.00	0.00	0.00
.	1.25	0.00	0.00	0.00	0.00
.	0.90	0.00	0.00	0.01	0.00
.	0.90	0.00	0.00	0.00	0.00
.	1.68	0.00	0.00	0.00	0.18
.	1.26	0.00	0.00	0.00	0.00
	0.92	0.01	0.04	0.00	0.02
Connections of R	0.90	0.03	0.00	0.00	0.00
Nodes. ⟶	1.35	0.00	0.00	0.00	0.00

Note: Connections from reservoir nodes shown in rows; the output nodes shown on the column

The next step is to add another input item to the sequence. So, first input item (A) was learned. Now item (B) is given as input, assuming item (A) has been learned. The tables below show the activation of the output nodes after reactivation with input item (A).

Table 4 shows the evolution of the activation of output nodes over time for all output nodes. This is given by the time parameter and simulates the development of the output nodes per given time step (shown on the rows). The columns show the activation for the output nodes when the input item (A) is given for reactivation of the sequence after learning the output.

The output node representing item (B) is learned and reactivated when given input (A). The input node (A) activates the reservoir nodes, which in turn activate the output nodes via the learned connections. As with the previous simulation, there is no activation for any of the output nodes in the first two-time steps. After three times steps there is an activation of 0.76 for the output node representing item (B), but no activation for the node representing item (A). Item (B)

has been learned, but there are no connections for item (A). The sequence (A, B) was not learned. Previous connections that were learned for item (A) are erased when item (B) is learned.

Table 4

Evolution of the activation of the output nodes over time, when input node A and B are learned, and input A is reactivated

	Output nodes:				
	A	**B**	**C**	**D**	**E**
Time step^0 ⟶	0.00	0.00	0.00	0.00	0.00
.	0.00	0.00	0.00	0.00	0.00
.	0.00	0.76	0.00	0.00	0.00
.	0.01	0.94	0.00	0.00	0.00
.	0.01	0.98	0.00	0.00	0.01
.	0.02	0.99	0.00	0.00	0.01
.	0.02	0.99	0.00	0.00	0.01
.	0.02	0.99	0.00	0.00	0.01
.	0.02	0.99	0.00	0.00	0.01
Time step^n ⟶	0.02	0.99	0.00	0.00	0.01

Note: time is shown in rows; the output nodes shown on the column

If the items (A, B) were learned, the sequence should be reactivated when item (A) is given again. The input item (A) activates the reservoir nodes associated with this item, which in turn activate the output nodes through their learned connections. This can be visualized by looking at the connections from the reservoir nodes to the output nodes after learning. Table 5 shows the output connections after learning. The rows show the connections from the reservoir to the output nodes (columns). The connections to output node representing item (B) are high. There are no connections formed for item (A), which indicates further that the connections previously made for item (A) have been forgotten/erased.

Table 5

Output connections after learning when input node A and B are learned

	Output nodes:				
	A	**B**	**C**	**D**	**E**
Connections of R	0.00	1.08	0.00	0.00	0.00
Nodes ⟶	0.00	1.08	0.00	0.00	0.00
.	0.00	1.08	0.00	0.00	0.00
.	0.00	1.08	0.00	0.00	0.00
.	0.00	1.08	0.00	0.00	0.00
.	0.16	1.08	0.00	0.00	0.06
.	0.00	1.08	0.00	0.00	0.00
	0.00	1.08	0.01	0.00	0.00
Connections of R	0.00	1.11	0.00	0.00	0.00
Nodes. ⟶	0.00	1.11	0.00	0.00	0.00

Note: Connections from reservoir nodes shown in rows; the output nodes shown on the column

Hence, it is possible to learn one individual item, but when a new item is learned the previously learned item is lost. Therefore, it is not possible to learn a sequence in a sequential manner. Only the results for the items (A, B) are shown here to illustrate the issue, for other input nodes the results are similar, previously learned items are erased when new items are learned.

3.3 Discussion

The results of the simulation showed that learning a single item is possible, but when a new item is introduced, earlier items are forgotten. Hence, a sequence cannot be learned in a sequential manner. When the items are learned the connection weights are adapted, but when a new item is learned it washes away the connections of the previously learned items. This can be explained by catastrophic interference. Catastrophic interference, also known as catastrophic forgetting, is a well-known problem in neural networks. It is described as the tendency of neural networks to forget old items as new items are learned (Endress & Szabó, 2020; McCloskey & Cohen, 1989).

This makes it difficult to train networks in a sequential manner. Catastrophic interference occurs especially when the network is trained sequentially, because the weights in the network used for one task get changed to meet the objectives for the second task (Kirkpatrick et al., 2017). Being able to learn data in a sequential manner is a necessary condition to model human cognition. While human working memory does show similar behaviour in terms of capacity limitations, they do not reflect the level of catastrophic interference seen in neural networks (Endress & Szabó, 2020).

Catastrophic interference occurs with a reservoir network because the standard reservoir computing approach does not actually use sequential learning. All sequential relations are stored in the reservoir and then presented simultaneously for learning. It was initially assumed that, because the reservoir computing approach is capable of learning sequences, it would be capable to do this in a sequential manner. But the first versions of the network (Experiment 1) showed that this was not the case. This assumption came from prior studies on sequence learning using RC, such as the paper by Hinaut and Dominey (2013) where they demonstrated that a recurrent neural network is capable of decoding grammatical structure from sentences in real time, generating a predictive representation of the sentence meaning. However, their system learns the sequences inside the reservoir by collecting all information from the reservoir and then simultaneously present them to the learning procedure to produce the desired output. This way the sequences are not learned sequentially. This also gives insight into why the connections in the reservoir are fixed. They give a set of sequences which then can be presented as a single pattern.

While the network did learn the grammatical sequences presented, it did not do this in a sequential manner, learning sequences step by step as discussed in the introduction. This does not mean that their study did not achieve what it was designed to do. Rather it shows there is a distinction between learning sequences and sequential learning, which casts doubt on the cognitive ability of the system, as it does not learn step by step. To address this issue, the learning should take place in the reservoir, rather than in the readout to the output, as the sequential nature of a reservoir network lies in the reservoir. In the next experiment, we look at learning internally in the reservoir, instead of learning the output connections.

4.0 Experiment 2: Changing the role of learning

In the current approach to reservoir computing the learning takes places outside of the reservoir, from the reservoir nodes to the output nodes. Sequences are fixed and are represented in the reservoir. These sequences can then be presented to a learning algorithm simultaneously and learned. The results of Experiment 1 showed that when new items are introduced to the network, previously learned items are being forgotten. Learning a sequence in a sequential manner was not possible. This raises the question why this effect did not occur in the standard reservoir approach. This can be explained by looking at how the sequences have been learned. By storing all the information within the reservoir and then presenting these simultaneously to the learning algorithm, it is as if the system learns a spatially organized pattern. While this approach is capable of solving many different problems, it does not yet simulate accurately how sequential learning occurs in human cognition.

How can a basic sequence be learned in a sequential manner? To solve this issue a new approach is introduced, which changes the role of learning. The sequential nature of the network lies within the reservoir and its random like connections. It is already known from Experiment 1 that it is possible to learn an individual item, based on connections from the input nodes to the reservoir and from the reservoir to the output nodes. So, instead of learning the connections from the fixed reservoir to the output nodes, we will now use fixed connections from the input to the reservoir nodes and fixed connections from the reservoir nodes to the output nodes. Sequential learning of a sequence will now be inside the reservoir. That is, the focus of learning is now inside the reservoir instead from the reservoir to the output nodes. The aim is to achieve sequential learning, which is not achieved by current methods.

The aim of Experiment 2 is to learn a basic sequence consisting of five items. The input items are represented by letters (A, B, C, D, E), as in Experiment 1. Each letter represents an input item, which will form the basic sequence. The output nodes are also those as in Experiment 1, but now with fixed selective connections from the reservoir nodes. So, when an item (e.g., C) is presented, it activates the reservoir node to which it is connected, and this activates the output node representing (C). This occurs for each item in the sequence. So, each item in the sequence is represented by a selective node in the reservoir. The aim is now to learn the sequence of items presented to the reservoir by modifying the connections between the reservoir nodes.

To check if the sequence was learned only the starting item of a sequence is represented again. If learning occurred, it should allow for the reactivation of the output nodes for the entire sequence of all five items. To test the performance of the network, several simulations were performed. In particular, the influence of the parameter "repeat" was tested, which controls how many times the sequence is presented and how many learning iterations the network goes through before the sequence gets reactivated entirely by only presenting its first item.

4.1 Method

Network implementation

The reservoir was built using the programming language Python version 3.6 using the NumPy library. The relevant code can be found in Appendix B.

The basic structure and procedure are similar to the previous version of the network. The network was made by creating connection matrices between layers and within the reservoir. The network consists of three layers, the input, the reservoir, and the output layer. Connections in the reservoir are possible for each node to every other node expect to itself. The model architecture is used to simulate the presentation and learning of a basic sequence inside the reservoir (R).

The main difference is that learning now takes place in the reservoir and not between reservoir and output. This way the network can learn the items of the sequence step by step instead of simultaneously learning a sequence as a single pattern. All connections from input nodes to reservoir nodes and from the reservoir nodes to the output nodes are fixed, in such a way that each item presented to the reservoir activates the output node that represents it. By adapting the connections within the reservoir learning occurs. The network does this in the following way. One node in the reservoir is randomly selected to represent a single item in the sequence. Hence, each item in the sequence is represented by a single and unique node in the reservoir. The connections from the input node to a specific node in the reservoir, and then to the output node representing the same item are fixed. Therefore, when input node (A) is activated, it will activate the output node (A), via the reservoir node for (A).

The connections between the reservoir nodes are initially random. They are modified based on the activations of the reservoir nodes that result from the presentation of the input sequence. The learning rule used is Hebbian learning as discussed earlier. The connection weights between two nodes are dependent on their mutual activation. The connection between an

activated node at time “t” (the “pre-synaptic” node) and the node active at “t+1” (the “post-synaptic” node) are modified (strengthened). Through this, it is ensured that when the node originally active at time t is reactivated, it will also reactivate the node that was originally active at time “t+1”. This way the sequence is learned. The network learns the items (A, B, C, D, E) that make up the sequence. If learning occurred, the entire sequence is reproduced when only the starting item is represented again.

Experiment 2

The second experiment (simulation) simulates the learning and reactivation of a basic sequence consisting of five items (A, B, C, D, E). The network consists of 5 input nodes, 10 reservoir nodes and 5 output nodes. Sparsity was set to 0.0. To test network performance the influence of the parameter “repeat” on the reactivation of the basic sequence was investigated. The “repeat” parameter controls how many times the items are presented and how many learning iterations the network goes through before the sequence is reactivated by presenting the starting item of the sequence.

4.2 Results

The repeat parameter, which controls learning iterations, was varied from small (1-10) to larger (20-100 in increments of 10). Varying the amount of learning iterations shows how many repetitions are needed to learn the desired sequence. Figure 3 shows the reactivation values of the output nodes with varied amount of learning iterations from 1-10. On the X-axis the individual items of the sequence are shown for items (A, B, C, D, E). The y-axis shows the activation of the output nodes, after the predetermined number of learning iterations. If learning occurred, the entire sequence should be reactivated when only input item (A), the starting item of the sequence, is presented. When learning failed, the output nodes are not reactivated and the activations for the output nodes are 0. For the learning iterations from 1-10, reactivation of the entire sequence was not possible. When the starting item (A) is presented again, only the output node representing item (A) is reactivated. The rest of the output nodes, representing items (B, C, D, E) are not activated. Less than 10 learning iterations do not seem sufficient to learn the basic sequence.

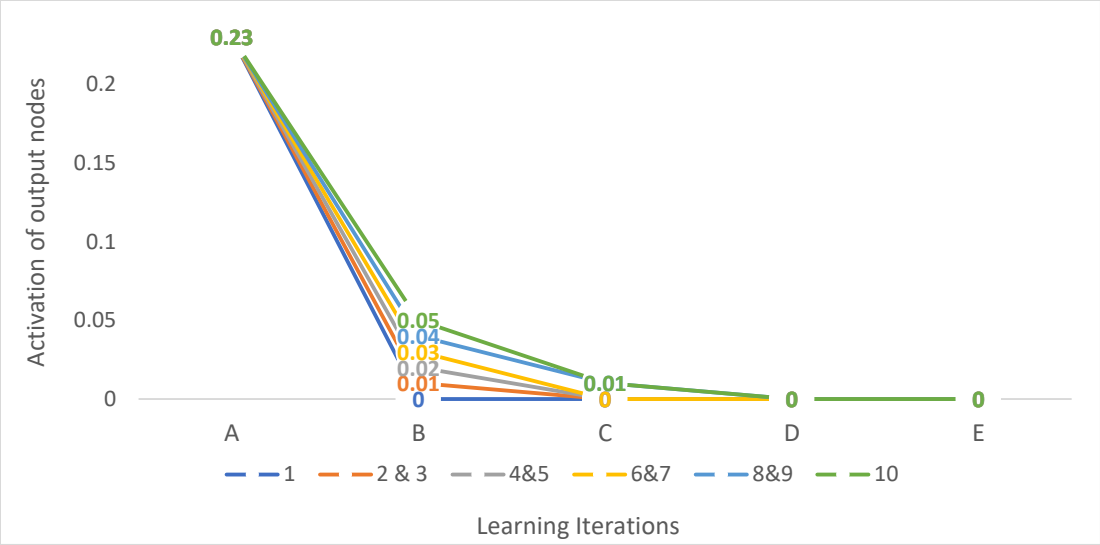

Figure 3 Reactivation values of output nodes with learning iterations 1–10

The next step was to increase the repeat parameter from 10-100 in increments of 10. Figure 4 shows the reactivation values of the output nodes for learning iterations from 10–100. On the X-axis the individual items of the sequence are shown for items (A, B, C, D, E). The y-axis shows the activation of the output nodes, after the predetermined number of learning iterations. The repeat parameter was increased to find the optimal level of iterations for learning to occur. After 10 iterations, shown in blue, the reactivation of the sequence was not possible. For 20 iterations, shown in orange, the sequence is learned, although the activation of the output nodes is low, for node D = 0.04 and node E = 0.03. For 30 iterations, shown in grey, the activation of the output nodes representing the items (A, B, C, D, E) becomes higher. The sequence was successfully learned and reactivated by the stating item of the sequence. The activation values of the output nodes become higher for 40 iterations. The activation values do not change much more, when learning iterations are increased from 50-100 learning iterations. The sequence is learned when at least 30 learning iterations are applied.

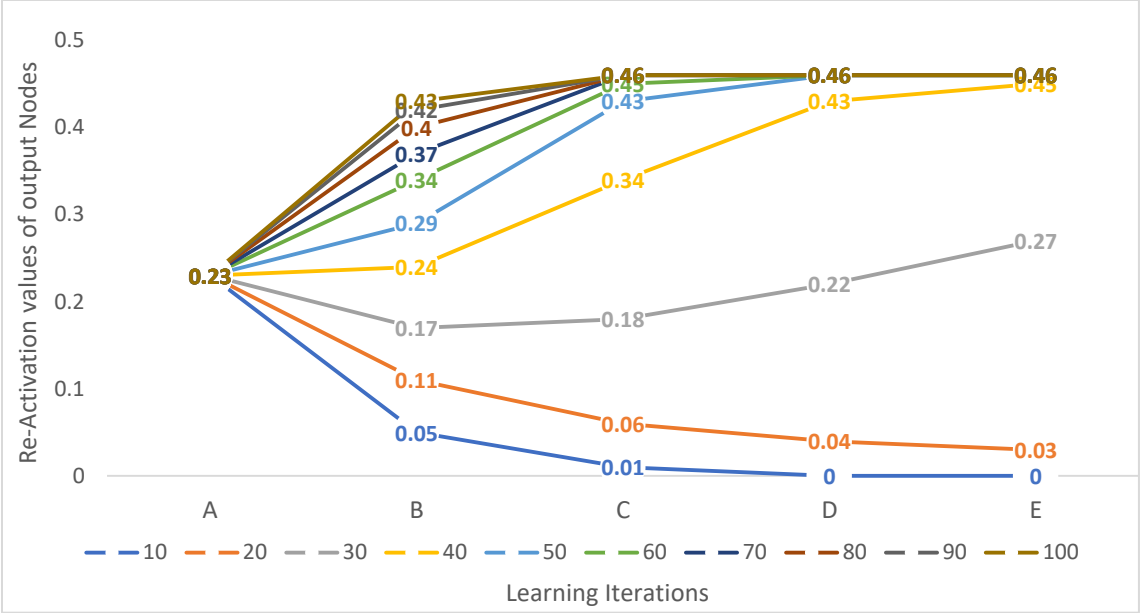

Figure 4 Reactivation values of output nodes with learning iterations 20-100 in steps of 10

4.3 Discussion

The aim of Experiment 2 was to learn a basic sequence consisting of five items. The input items are represented by letters (A, B, C, D, E). Each letter represents an input item, to form the basic sequence. To test the performance of the network the influence of the network parameter "repeat" was investigated. The sequence was successfully learned, when at least 30 learning iterations were applied. Too few iterations showed that the output nodes were not reactivated, therefore the sequence was not learned. When learning iterations were increased output nodes activation increased but levelled after a certain number of iterations.

The results show that strong connections between the items can be formed and that when learning is placed inside the reservoir it is possible to learn a sequence sequentially. However, the current network is only capable of learning one basic sequence, without repetitions of items. The next network version aims to build a basis for a network that is capable of learning a sequence with repeating items in a sequence.

5.0 Experiment 3: Effects of sparsity

The previous network version showed that it is possible to learn one basic sequence consisting of five items in a sequential manner, when learning was placed inside the reservoir. The previous network described in Section 4.0, was only capable of learning one unique sequence with no repetitions of items. When repeating items were introduced, the sequence was not learned. Experiment 3 aims to begin to build a network that is capable of learning sequences with reoccurring items by introducing cluster nodes. That means, multiple nodes can represent a single item in the sequence. To this end, Experiment 3 focuses on the effects of sparsity on learning and reactivation of the output nodes. Sparsity gives the chance of having non-zero connections between the nodes in the reservoir. A high sparsity results in fewer connections because the threshold for making a connection is higher. The maximum sparsity is 1.0, which gives zero connections. The minimum sparsity is 0.0, in which each reservoir node is connected to all other reservoir nodes (reservoir nodes are not connected to themselves).

5.1 Method

Network implementation

The reservoir was built using the programming language Python version 3.6 using the NumPy library. The relevant code can be found in Appendix C.

The basic network architecture and procedure is similar with the network version described in Section 4.0 . Only the differences are outlined here. In the previous network each item was represented by one unique reservoir node. Now the reservoir allows for a cluster of multiple reservoir nodes that represent one item in a sequence. The size of the cluster can be varied but is the same size for each item in the reservoir. For example, with five items, when the reservoir size is set to 10 and cluster size to 2, each item has two nodes representing that item (cluster size * number of items ≤ reservoir size).

Experiment 3

The third experiment (simulation) simulates the learning and reactivation of a basic sequence consisting of five items (A, B, C, D, E). The simulations to test the performance of the networks examines the effect of sparsity on the network. The network consists of 5 input nodes,

10 reservoir nodes and 5 output nodes. The cluster size was set to 2, which means there are 2 reservoir nodes associated with each item. The parameter “repeat” was set to 30, based on the results of Experiment 2. Sparsity was set to 0.0; 0.2; 0.5; 0.9 for the simulations.

Additionally, it was tested if it is possible to learn a sequence with one repeating item, consisting of the items (A, B, A, C, D). Five input items were given, the reservoir size was 10, with a cluster size of 2. The parameter “repeat” was set to 30 and sparsity was set to 0.0.

5.2 Results

The simulations to test the performance of the network examine the effect of sparsity on the network. The simulation showed that for a basic sequence with five items (A, B, C, D, E) lower level of sparsity yields better results for learning. As sparsity increases, the amount of possible reservoir node connections decreases, so does the success rate for learning the sequence. Figure 5 shows the reactivation values of the output nodes with varying sparsity for the sequence (A, B, C, D, E). It is expected that for higher sparsity, the sequence is not learned, because it will allow fewer connections between the reservoir nodes. The sequence is successfully learned for sparsity of 0.0 and 0.2. For higher sparsity learning of the sequence fails, from a sparsity of 0.5 only parts of the sequence are learned. For higher sparsity learning the sequence is not possible, which is expected as the number of possible connections between reservoir nodes decreases.

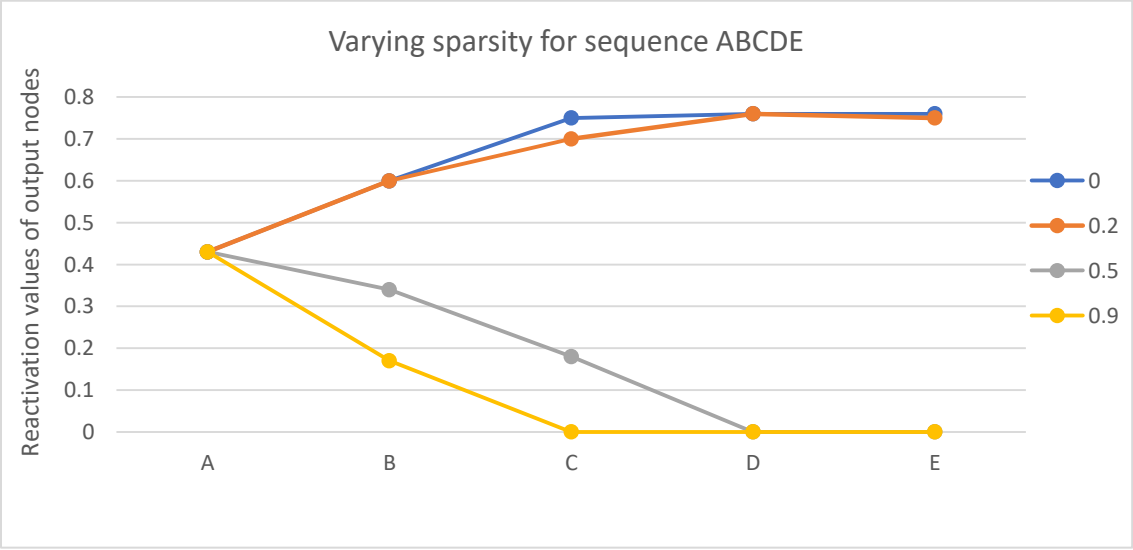

Figure 5 Reactivation values of output nodes with varying sparsity for sequence ABCDE

The item clusters in the reservoir provide multiple nodes to represent one item. Hence, this would allow the repetition of an item in a sequence. Figure 6 shows the reactivation values for the output nodes for the sequence (A, B, A, C, D). These results show that the current version of the network was not yet capable of learning a sequence with repetitions of items. The network does not distinguish between the presentation of (A) the first time and the presentation of (A) the second time. When the starting item of the sequence (A) is presented for the reactivation of the sequence, it activates the output nodes for items (B) and (C) simultaneously, because both are associated in the reservoir with the nodes activated by A. In turn, the reservoir node for item (B) also activates output (A) again, as a connection between (B) and (A) has been learned in the reservoir. The reservoir node for (C) activates the output node for (D) because the connection between (C) and (D) has been learned in the reservoir. In turn, the reoccurring activation of item (A) activates items (B) and (C) again through their learned connections in the reservoir, resulting in the repetitive activation of (A) and (C).

Hence the network did not seem to use the cluster nodes associated with item (A) in the reservoir to distinguish between the first and the second presentation of (A). Instead, it used the both clusters nodes for each presentation of (A). Therefore, the sequence is not learned as desired.

	Items (Output Nodes)				
	A	**B**	**A**	**C**	**D**
Sequence ABACD	0.43	0	0	0	0
	0	0.76	0	0.76	0
	0.76	0	0	0	0.76
	0	0.76	0	0.76	0
	0.76	0	0	0	0.76

Figure 6 Reactivation values of output nodes for sequence ABACD

5.3 Discussion

Experiment 3 simulates the effects of sparsity on learning and reactivation of a basic sequence consisting of five items (A, B, C, D, E). The reactivation values of the output nodes were best for lower levels of sparsity set to either 0.0 or 0.2. The reactivation values of the output nodes for

lower sparsity of 0.0 and 0.2 show that the connections are stable for all items. A lower level of sparsity allows for more connections within the reservoir and yields a better network performance for learning.

Cluster nodes were introduced to allow for repetitions of items. However, this network version did not make adequate use of that, as the reservoir nodes in the cluster formed connections with each item associated with the repeated item (A). When an item was represented again it also activates all other nodes connected to the item in the sequence and therefore does not give the desired output. The sequential order is lost, because there are multiple associations between reservoir nodes, which explains why there is a difficulty in producing a sequence where items are repeated.

The following network version aims to address this issue to allow learning of a sequence with repeating items.

6.0 Experiment 4: Reservoir with inhibition of return

The results presented in Section 5.0 showed that it was not possible to learn a sequence with repeating items. The following network aims to address this issue. To learn a sequence that contains repetitions of items and also longer sequences an inhibition of return like mechanism has been implemented. Inhibition of return has been widely studied in the domain of visual perception and selective attention and describes the suppression of processing of previously encountered stimuli (List, 2007). Inhibition of return encourages the orientation to new objects and supresses the attention to objects which have already been encountered. It is used here to address the issue of serial order by inhibiting the reuse of an item node in a cluster that has previously been used in the sequence. Instead, a new cluster node for an item is chosen in the reservoir when that item is presented again in the sequence.

The network described in Section 5.0 (Experiment 3) already contained a cluster of reservoir nodes corresponding to a single item. To learn sequences with repetitions of items an inhibition of return like mechanism was assumed, to make sure that when an item is presented again it will activate another reservoir node in the cluster associated with that item. In this way, a sequence with item repetitions can be learned as a unique sequence, because a new reservoir node is selected for each new presentation of an item. To test network performance, two sub experiments are performed. First, the success or failure of reactivation for sequences of different lengths is examined (Experiment 4.1). The second simulation (Experiment 4.2) looks again at the effects of sparsity on the network performance.

6.1 Method

Network implementation

The reservoir was built using the programming language Python version 3.6 using the NumPy library. The relevant code can be found in Appendix C.

The basic network architecture and procedure is similar with the network version described in Section 5.0. Only the differences are outlined here. The reservoir allows for multiple reservoir nodes that represent one item in a sequence. These are represented in a cluster of nodes inside the reservoir to represent the items. In the previous version the network did not make use of the cluster nodes as was intended. Therefore, an inhibition of return like mechanism was implemented to prevent the use of more than one reservoir node for a particular item. It operates

by randomly selecting one reservoir node from the cluster, when its item is presented. The use of that cluster node is prevented for any new representation of that item in a sequence. So, when an item is presented again it will make use of another cluster node associated with that item. Now a sequence containing repetitions of items should be learned as desired, as a new node in the reservoir is selected for each new presentation of an item.

Experiment 4.1

Experiment 4.1 (simulation) simulates the learning and reactivation of a basic sequence consisting of five items (A, B, C, D, E). The length of the sequence was varied and is tested with reoccurring items in the sequence. The simulations first test the failure/success of reactivation and learning of a sequence. The size of the sequences is varied from 5 to 22 items. Reservoir size is also adjusted to accommodate for more input items. The cluster size is adjusted relative to the reservoir size and amount of input items. The parameter "repeat" was set to 30, based on the results of Experiment 2. Sparsity was set to 0.0 for the simulation.

Experiment 4.2

Experiment 4.2 (simulation) simulates the learning and reactivation of a basic sequence consisting of five items (A, B, C, D, E). The length of the sequence was varied and is tested with reoccurring items in the sequence. The simulations to test the performance of the networks examines the effect of sparsity on the network. The number of items in a sequence varied from 7; 10 & 20. Reservoir size is also adjusted to accommodate for more input items. The cluster size is adjusted relative to the reservoir size and amount of input items. The parameter "repeat" was set to 30, based on the results of Experiment 2. Sparsity was set to 0.0; 0.2; 0.5 & 0.9 for the simulations.

6.2 Results

To see if the network can now learn longer sequences, different lengths of sequences with repetitions of items have been simulated (Experiment 4.1). A second simulation (Experiment 4.2) looks at the effect of sparsity on the learning and reactivation success for sequences of different length.

6.2.1 Experiment 4.1

Experiment 4.1 checks the success/failure of reactivation of the output nodes for sequences of different length with repetition of items. Table 6 below shows the success of reactivation of output nodes for sequences of different length with repetitions of items. In all simulations the sparsity was set to 0.0. For increasing sequence length, the reservoir size and the cluster size were adjusted.

Table 6

Success/Failure of reactivation of output nodes from sequences of different length with repetition of items

Number of items in sequence	Reservoir size	Cluster size	Success/Failure of reactivation
5	10	2	Success
6	10	2	Success
7	10	2	Success
8	10	2	Success
9	10	2	Success
10	10	2	Success
11	10	2	Failure
11	20	4	Success
12	20	4	Success
13	20	4	Success
14	20	4	Success
15	20	4	Success
20	20	4	Success
22	20	4	Failure
22	40	8	Success

Note: Reservoir and cluster size are adjusted for the sequence length. Sparsity was set to 0.0 for all simulation runs

Simulations with a sequence consisting of 5 items (A, B, C, D, E) were already performed in earlier simulations. The same sequence was used as the starting sequence and more items were added, containing any of the 5 items previously used. The sequences were learned up until the sequence reached a length of 11, then the reservoir size and cluster size needed adjusting, to allow for more items. Simulations were performed with sequences from 11 to 15 items. A sequence of length 20 and 22 were tested after the reservoir size and cluster size were adjusted. A sequence containing more items needed more time to compute due to larger reservoir size and items to be learned.

6.2.2 Experiment 4.2

Experiment 4.2 tests the effect of sparsity of connections in the reservoir on learning and reactivation for sequences of different lengths with repetition of items. Figure 7, 8 and 9 show these results for different sequence lengths of 7, 10 and 20.

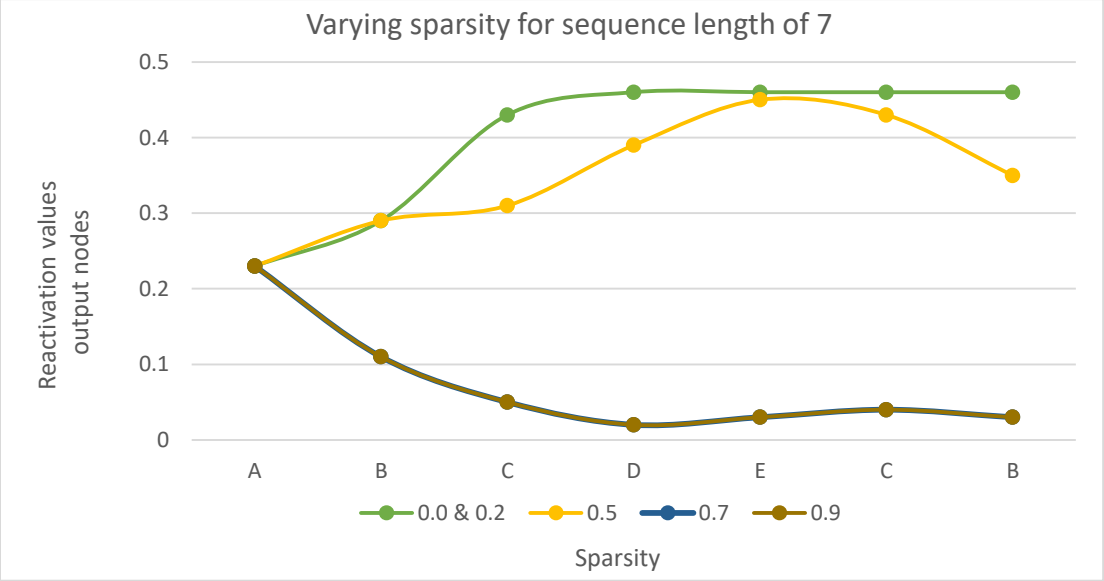

Figure 7 Reactivation values of output nodes with varying sparsity for sequence of length 7 with reoccurring items

Figure 7 shows the reactivation values of the output nodes varying sparsity (0.0, 0.2, 0.5, 0.9) for a sequence length of 7. The graph shows the nodes on the x-axis and the activation values of the output nodes on the y-axis. The coloured lines show the varying sparsity levels. For

higher sparsity of 0.7 and 0.9 the reactivation values of the output nodes become low and the sequence is not learned. The values for sparsity 0.0 and 0.2 are the same and show that the output nodes are reactivated. For sparsity of 0.5 the sequence is learned but not as optimally as with a lower sparsity.

Similar behaviour can be observed for sequences of length 10 as seen in Figure 8. For sparsity of 0.0 and 0.2 the sequences are learned, and reactivation values are high. For a sparsity of 0.5 the sequence is also learned, but reactivation values becoming weaker for later items in the sequence. For higher sparsity of 0.7 and 0.9, the sequence is not learned, and the output nodes were not reactivated.

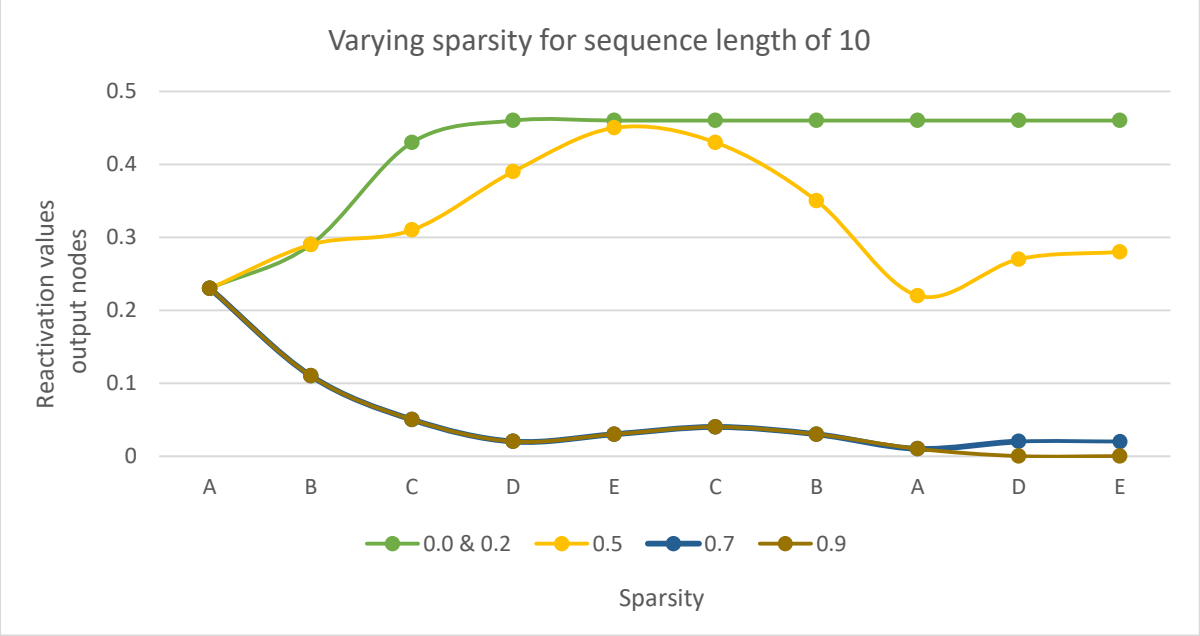

Figure 8 Reactivation values of output nodes with varying sparsity for sequence of length 10 with reoccurring items

The last sequence length to be simulated was of length 20. Longer sequences were not considered here, as the previous simulation showed similar trends; hence it is assumed that longer sequences will react similarly in terms of the effect of the sparsity on the connections in the reservoir.

Figure 9 shows the reactivation values of the output nodes with varying sparsity for a sequence of length 20 with reoccurring items. As seen with shorter sequences as well, the

sequence is learned only with lower sparsity of 0.0 and 0.2. When the sparsity is high (0.7 & 0.9) the sequence is not learned, and the nodes are not reactivated. For the longer sequence a sparsity of 0.5 does not seem sufficient, as after a few items the connections are not reactivated or only with a weak activation. Although the activation values increase again for later items. This is likely to happen, as the beginning items are the same as the last items in this sequence, so the network does remember the starting sequence again, through the previously learned connections.

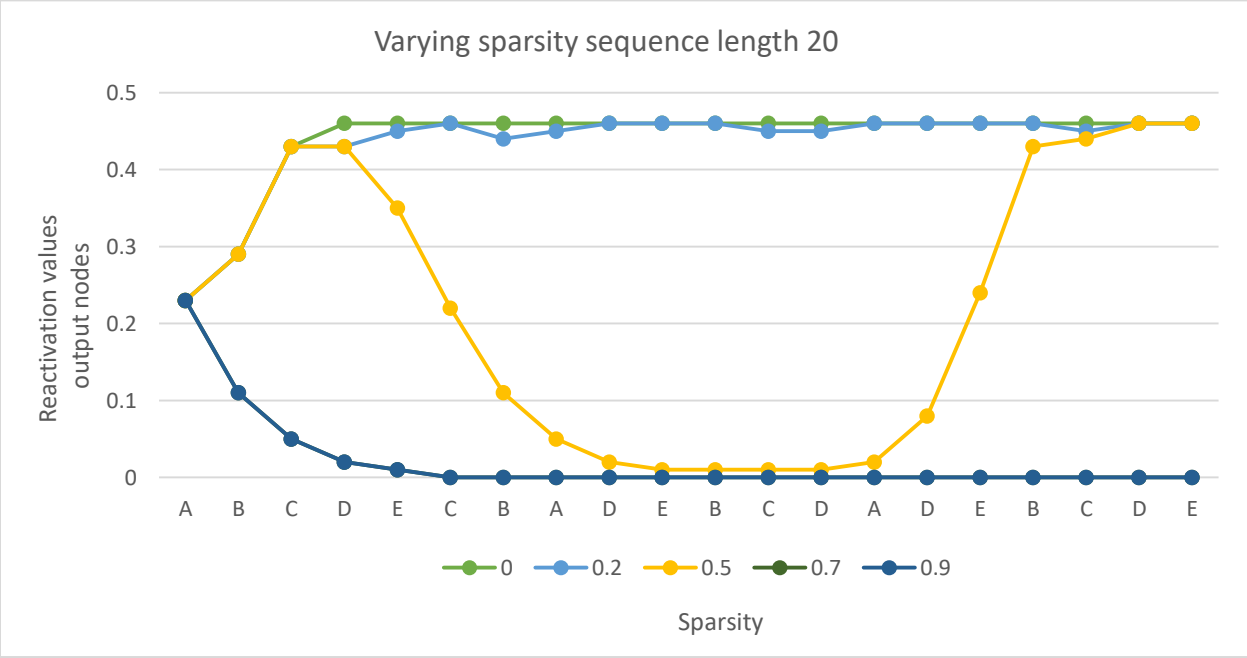

Figure 9 Reactivation values of output nodes with varying sparsity for sequence of length 20 with reoccurring items for all sequence lengths it can be said that a lower sparsity is needed for the sequence to be learned sufficiently

6.3 Discussion

The network version described uses an inhibition of return like mechanism on the reservoir nodes to allow for the reoccurrence of items in the sequence. It was possible to learn longer sequences with repetition of items. These can now be learned as a unique sequence as a new reservoir node is selected for each new presentation of an item. To test network performance, the success or failure of reactivation for sequences of different lengths was examined as well as the effects of sparsity on the network.

It was possible to learn longer sequences up to a length of 20 items, when the reservoir size is adjusted to a larger number of items. The network concatenates the items in the sequence and so can learn longer sequences. Performance of the network was also tested by looking at the effect of sparsity on the network. The different levels of sparsity were tested for sequences of different lengths. The reactivation values of the output nodes were best for lower levels of sparsity, set to either 0.0 or 0.2. The reactivation values of the output nodes for lower sparsity of 0.0 and 0.2 show that the connections are stable for all items. This was to be expected as previous results discussed in earlier sections showed similar results in regard to the sparsity of connections. The reactivation values remained stable for the entire sequence when lower sparsity was used.

The next step is to simulate learning more than one sequence. Learning a single sequence with repeating items builds a basis to learn more complex sets of sequences.

7.0 Experiment 5: Learning and reactivation of multiple sequences in the reservoir

The last simulations in this project simulate learning and reactivation of multiple sequences in the reservoir. The sequences consist of 5 items (A, B, C, D, E) like in the previous versions. The structure of the network is different. There is no input layer anymore, the sequence does not activate the input nodes, which then activate the reservoir nodes. Instead, the reservoir nodes are activated directly, to allow for easy control over the nodes. The aim is to learn and reactivate multiple sequences in the reservoir. That is, it will be investigated if more sequences can be stored in a single RC network. Furthermore, it will be investigated how the internal structure of a reservoir computing network affects the number of sequences that can be stored in a RC network. Lastly, it will be investigated how the sequences can be distinguished in a reservoir, so that a confusion between sequences in recall be prevented.

7.1 Method

Network implementation

The last version of the network simulates learning and reactivation of multiple sequences in the reservoir. The reservoir was built using the programming language Python version 3.6 using the NumPy library. The relevant code can be found in Appendix D.

The basic network architecture is similar with the network version described in the previous section. Only the differences are outlined here. The sequences consist of 5 items: A, B, C, D, E. Items are represented in the network by numbers: A=0, B=1, C=2, D=3, E=4. There is no input layer anymore. The sequence does not activate the input nodes, which then activate the reservoir nodes. Instead, the reservoir nodes are activated directly, to allow for easy control over the cluster nodes. This network is capable of learning multiple sequences of different lengths with repeating items.

Experiment 5.1

The first sub experiment (5.1) looks at the success/failure of the reactivation of the output nodes for two and five individual sequences with different lengths. The length of the sequences was varied and tested with reoccurring items in the sequence. The simulations first test the failure/success of reactivation and learning of two individual sequences and then of five

individual sequences. The number of input items is varied. Reservoir size is also adjusted to accommodate for more input items. The cluster size is adjusted relative to the reservoir size and amount of input items. The parameter "repeat" was set to 30, based on the results of Experiment 2. Sparsity was set to 0.0 for the simulation.

Experiment 5.2

The second sub experiment (5.2) looks at the effect of varying sparsity of connections in the reservoir on the reactivation in the reservoir for multiple sequences of different lengths with repetition of items. The simulations to test the performance of the networks examines the effect of sparsity on the network. The simulations were made for two and five distinct sequences. Reservoir size is also adjusted to accommodate for more input items. The cluster size is adjusted relative to the reservoir size and amount of input items. The parameter "repeat" was set to 30, based on the results of Experiment 2. Sparsity was set to 0.0; 0.2; 0.5 & 0.9 for the simulations.

7.2 Results

To test the network performance, two experiments were performed. The first one simulates a set of sequences with repetitions of items (Experiment 5.1). The second simulations (Experiment 5.2) performed with this network looked at the effect of varying sparsity on the reactivation values of the output nodes for multiple sequences of different lengths with repetition of items.

7.2.1 Experiment 5.1

To test the network performance, multiple sequences with repetitions of items have been simulated. Table 7 shows the results of the simulation of two sequences with repetitions of items. Two individual sequences were learned, the length of the sequences was varied. To check if the sequence was learned, the reactivation values of the output nodes were checked.

Table 7 shows the results of 2 sequences learned with increasing sequence lengths with repetition of items, each sequence starting with a different starting item. The two sequences were distinct from each other in terms of serial structure, the sequences had a different order of items. The sparsity for all sequences was set to 0.0. To check if the sequence was learned successfully, the reactivation of the output nodes sequence was examined. It was possible to learn 2 sequences with a sequence length of 5. Then the sequence lengths are increased by a step of one until the

sequence length reached 10. The network learned 2 sequences of length 10. To check if longer sequences are learned a sequence length of 20 items was simulated as well. Although the computational time increased, which is to be expected, the network was able to learn the 2 sequences of length 20.

Table 7

Success/Failure of activation of output nodes after reactivation of two sequences

Individual sequences learned	Sequence length	Sparsity	Success/Failure
2	5	0.0	Success
	6	0.0	Success
	7	0.0	Success
	8	0.0	Success
	9	0.0	Success
	10	0.0	Success
	20	0.0	Success

Note: Simulation results of sequences with repetitions of items. 2 individual sequences learned with varying sequence length

Table 8 shows the results of 5 sequences learned with increasing sequence lengths with repetition of items, each beginning with a different starting item. The five sequences were distinct from each other in terms of serial structure, the sequences had a different order of items, as before with the two sequences. The sparsity for all sequences was set to 0.0. To check if the sequences were learned successfully, the reactivation of output nodes for each sequence was checked. It was possible to learn 5 sequences with an item length of 5, from there the sequence lengths was increased by a step of one until the sequence length reached 10. The network learned 5 sequences of length 10. To check if longer sequences are also being learned a sequence with a length of 20 items was simulated. Although the computational time increased, which is to be expected the network was able to learn the 5 sequences with 20 items.

Table 8

Success/Failure of activation of output nodes after reactivation of five sequences

Individual sequences learned	Sequence length	Sparsity	Success/Failure
5	5	0.0	Success
	6	0.0	Success
	7	0.0	Success
	8	0.0	Success
	9	0.0	Success
	10	0.0	Success
	20	0.0	Success

Note: Simulation results of sequences with repetitions of items. 5 individual sequences learned with varying sequence length

It was possible to simulate multiple sequences of different lengths with repetition of items when different starting nodes were used. However, when simulating sequences that start with the same starting node, the network does not distinguish between the two sequences. This is shown in Figure 10. Two sequences are stored, sequence 1 containing the items (D, B, C, A, E) and sequence 2 containing the items (D, C, D, A, B). They both start with item D, which is then used as input to reactivate the sequence after learning. Hence, when the node for the first node item (D) is activated, it activates all the reservoir nodes that represent D (as the inhibition of return mechanism is used only in the learning stage). At the second step, the nodes for both items (B) and (C) are active, that is, the second output node of each sequence is activated. The network reactivates the nodes simultaneously, there seems to be no distinction between the sequences. The next nodes to be activated are for items (C) and (D). So, the third output nodes of both sequences are activated. Again, both items are active, and the network does not distinguish between the two sequences. The fourth item in the sequences was the same (A). Here we can see that only the output node for item (A) is reactivated. The last nodes to be reactivated are for the items (B) and (E), which are also activated simultaneously. Hence, both sequences were learned, but the sequences are not distinguished between each other. The network did not distinguish between the repeated first item.

Nodes

A	B	C	D	E
0	0	0	0.23	0
0	0.3	0.19	0	0
0	0	0.45	0.43	0
0.76	0	0	0	0
0	0.46	0	0	0.46

Figure 10 Reactivation values of output nodes for sequence 1 (D, B, C, A, E) & sequence 2 (D, C, D, A, B)

7.2.2 Experiment 5.2

Experiment 5.2 looked at the effect of varying sparsity of the connections in the reservoir on the reactivation of the output nodes for multiple sequences of different lengths with repetition of items.

Figure 11 shows the effect of sparsity on the reactivation of the output nodes when 2 sequences are learned. Figure 11 shows the sparsity on the x-axis and the success (1) of reactivation or failure (0) of reactivation for the two individual learned sequences of length five. The blue line shows the reactivation of the output nodes for the first sequence. It shows that the sequence is learned for all sparsity parameter except 0.9. However, the second sequence, shown in orange is only learned with a sparsity of 0.0. The two sequences are only learned when sparsity is set to 0.0, as otherwise the sequence is only learned partially.

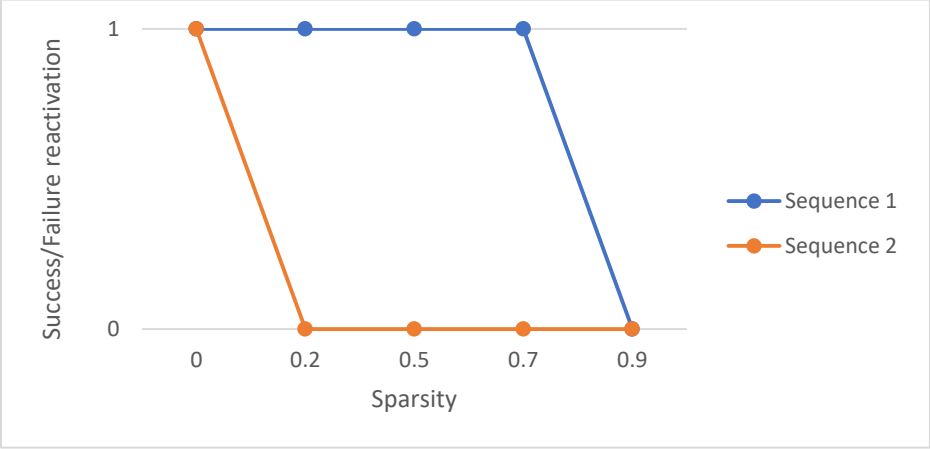

Figure 11 Effect of the sparsity of the connections in R on success of reactivation, when 2 sequences are learned

The next simulation looks at the effects on the reactivation of the output nodes when five different sequences of length 5 are learned. Figure 12 shows the effect of sparsity on the reactivation of the output nodes when 5 sequences are learned. The sparsity is shown on the x-axis and the success (1) or failure (0) of reactivation of the output nodes for the five sequences of with a of length five items on the y-axis.

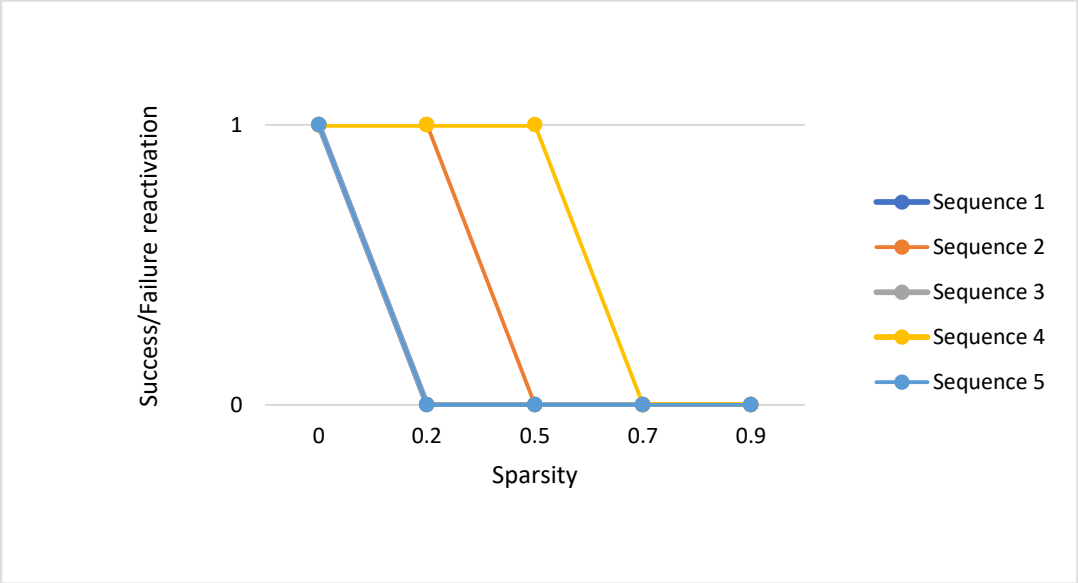

Figure 12 Effect of the sparsity of the connections in R on success of reactivation, when 5 sequences are learned

It was only possible to learn all five sequences when sparsity was set to 0.0. When sparsity increased, there was only partial success in learning the sequences. For a sparsity of 0.2 only 2 sequences were learned. When sparsity was set to 0.5 only one sequence was learned after reactivation of the sequence. For higher sparsity of 0.7 & 0.9 no sequence was reactivated, and the sequences were not learned.

7.3 Discussion

The last version of the network simulated learning and reactivation of multiple sequences in the reservoir. The sequences consist of 5 items (A, B, C, D, E) like in the previous versions. The network has no input layer anymore, the sequence is not activating the input nodes, which then activate the reservoir nodes. Instead, the reservoir nodes are activated directly, to allow for easy control over the nodes. The aim is to learn and reactivate multiple sequences in the reservoir.

This allowed investigating if more memory chunks (basic sequences) can be stored in a single RC network. The results show that it is possible to learn and reactivate more than one sequence in the reservoir. Simulations were run for two sequences with varied length and for five individual sequences with different length. The simulation was successful for sequences with different starting nodes in the reservoir, but not for sequences in which the first starting nodes in the reservoir are the same. This leaves the question of how the memory chunks that start with the same item can be separated. So, how can confusion between such sequences in recall be prevented? This topic could be subject for further research.

Furthermore, it was investigated how the internal structure of a reservoir computing network affects the number of memory chunks that can be stored in a RC network. The effects of different levels of sparsity on the reactivation of the output nodes were tested for sequences of different lengths. The reactivation of the output nodes was best for lower levels of sparsity set to either 0.0 or 0.2. The reactivation of the output nodes for lower sparsity of 0.0 and 0.2 shows that the connections are stable for all items. The connection strength remained stable for the entire sequence when lower sparsity was used. A high sparsity results in fewer connections because the threshold for making a connection is higher. For optimal network performance a low level of sparsity is needed to allow for optimal connection between the nodes.

8.0 General discussion

The aim of the current study was to investigate whether reservoir computing can be used to simulate sequential learning of basic sequences of items as found in a memory chunk, seen as an association between its items. In the beginning four sub questions have been asked to determine whether RC is a suitable method to investigate sequential learning of basic sequences.

The first question raised was how basic sequences can be represented in a reservoir computing network. The standard way of learning in reservoir computing seemed to be a suitable approach to learn basic sequences. However, the first simulations showed that the network was not capable of learning in a sequential manner. It can learn single items but cannot learn an entire sequence in a sequential manner step by step. It was not possible to learn new items in a sequence without losing previous connections made in the reservoir. This effect is due to catastrophic interference. This cast doubts on the cognitive ability of the current reservoir computing approach to learn sequences in a sequential manner. The role of learning was changed in the network to address this issue. Learning was introduced in the reservoir, rather than in the readout to the output, as the sequential nature of a reservoir network lies in the reservoir. Hence learning now occurred in the connections between the reservoir nodes, instead of in the connections between the reservoir nodes and the output nodes. This approach showed that it was possible to learn up to five distinct sequences with multiple reoccurring items.

The second research question stated if more sequences can be stored in a single RC network. The first network was only capable of learning one sequence without repetitions of items. Extensions of the network aimed to make the network more sophisticated, so it allowed for the learning and reactivation of multiple sequences in the reservoir. This allowed investigating if more sequences could be stored in a single RC network. Experiment 5 showed that with the model built in this project it was possible to learn 5 distinct sequences with multiple recurring items in a sequential manner. The results also showed that strong connections can be developed between the reservoir nodes that represent the items in a sequence.

Reber (2011) described requirements for a neural network to mimic sequential learning in the basal ganglia. He argues that current models need to be more complex and are not yet capable of learning different sequences in succession. Specifically, Reber (2011) describes an experiment in which participants learned 3 distinct sequences over a 2-day period and showed that all sequences can be retained equally well, he argues that this would not be possible for

current neural network approaches due to the effects of catastrophic interference. Hence, the current approaches are not yet capable of describing sequential learning as seen in the basal ganglia. Experiment 1 failed to learn a basic sequence, which is in line with argumentation of Reber (2011), namely, learning sequences in a sequential manner would not be possible due to catastrophic interference. However, the new approach introduced here shows that multiple sequences can be learned in succession. These results show a first approach to build a network that could mimic sequential learning as observed in the basal ganglia.

The third question raised was how does the internal structure of a RC network affect the number of sequences that can be stored in a single RC network? To answer this question, we looked at multiple parameters influencing network performance. The amount of learning iterations, sparsity, the size of the reservoir and of the cluster nodes all have influence on network performance. However, of particular interest are the effects of different levels of sparsity on the network in regard to the number of sequences that can be learned. Looking at the simulations there does seem to be an effect of sparsity on the length of a sequence that can be stored. This could give insight into why chunks are formed. So, why do we not just have very long associative sequences in memory but instead break them up into shorter ones? The reactivation of the output nodes was best for lower levels of sparsity set to either 0.0 or 0.2. The reactivation of the output nodes for lower sparsity of 0.0 and 0.2 shows that the connections are stable for all items. The connection strength remained stable for the entire sequence when lower sparsity was used, but not for higher sparsity levels, so when the connections were denser and the threshold to make a connection higher. For higher sparsity learning the sequence is not possible, the number of possible connections between reservoir nodes decreases which inhibits the formation of adequate connections in the reservoir.

The segmentation of a sequence into blocks or chunks makes it easier for the brain to retain and recall information (Ericcson, Chase, & Faloon, 1980). Hence, chunking is necessary to process large amounts of information and so make it more accessible / easier to remember. In human cognition these memory chunks are limited. The amount of information that can be processed and remembered at a given moment is limited. In the neural network there seemed to have been an effect of sparsity on the length of a sequence that can be stored. This is comparable to the limitations of human working memory as only a certain number of chunks or items in a sequence can be learned.

Lastly, the question was raised how can the sequences be separated? So, how can confusion between sequences in recall be prevented? The simulation in Experiment 5 were successful for sequences with different starting nodes, but not for sequences in which the first items in the sequence are the same. This leaves the question of how those chunks can be separated. The results showed that the current network was not yet capable of separating these sequences. However, it would be a necessary requirement to adequately mimic human sequential learning. As described earlier, already early in an infant's life, they are capable to learn the sequential structure of syllable sequences and detect new sequence of the same syllables (Saffran, Aslin, & Newport, 1996). Only when the sequences can be separated, we can build a model that accurately describes human sequential learning.

8.1 Limitations & further research

The presented network is a basic reservoir network to simulate learning and reactivation of sequences in the reservoir. The network was built to simulate only 5 items in the reservoir. In the future this could be expanded and made more sophisticated to use more items and process different sequences. The network could be used as a starting point to build a more sophisticated neural network to mimic human sequential learning.

An issue left unsolved in the current network concerned sequences that start with the same item. The simulation was successful for sequences with different starting items, but not for sequences with the same starting item. This leaves the question of how these sequences can be separated. So, how can confusion between such sequences in recall be prevented? This would currently prevent the network to learn more complex sequences.

Another suggestion for improvement lies in the internal structure of the network, when adjusting the cluster size, no checks are made if the cluster size is adequate. So, if (e.g.) too many of the same items are presented, the program will crash at some point. Sophistication of this issue could be subject of another project.

8.2 Conclusion

The initial results casted doubts on the cognitive ability of reservoir computing learning sequences in a sequential manner. However, a first step was made towards a better approach to simulate human sequential learning. By learning sequences within the reservoir instead of from

the reservoir to the output nodes a basic sequence could be learned in a sequential manner. The network simulated the learning and reactivation of multiple sequences within the reservoir.

9.0 References

Ahire, J. B. (2018). The Artificial Neural Networks Handbook: Part 4. Retrieved from https://medium.com/@jayeshbahire/the-artificial-neural-networks-handbook-part-4-d2087d1f583e

Bo, J., & Seidler, R. D. (2009). Visuospatial Working Memory Capacity Predicts the Organization of Acquired Explicit Motor Sequences. *Journal of Neurophysiology, 101*(6), 3116-3125. doi:10.1152/jn.00006.2009

Clegg, B. A., DiGirolamo, G. J., & Keele, S. W. (1998). Sequence learning. *Trends in Cognitive Sciences, 2*(8), 275-281. doi:10.1016/S1364-6613(98)01202-9

Conway, C. M. (2012). Sequential Learning. In N. M. Seel (Ed.), *Encyclopedia of the Sciences of Learning* (pp. 3047-3050). Boston, MA: Springer US.

Dominey, P. (2013). Recurrent temporal networks and language acquisition—from corticostriatal neurophysiology to reservoir computing. *Frontiers in Psychology, 4*(500). doi:10.3389/fpsyg.2013.00500

Ellis, N. C., & Sinclair, S. G. (1996). Working Memory in the Acquisition of Vocabulary and Syntax: Putting Language in Good Order. *The Quarterly Journal of Experimental Psychology Section A, 49*(1), 234-250. doi:10.1080/713755604

Endress, A. D., & Szabó, S. (2020). Sequential Presentation Protects Working Memory From Catastrophic Interference. *Cognitive Science, 44*(5), e12828. doi:10.1111/cogs.12828

Ericcson, K. A., Chase, W. G., & Faloon, S. (1980). Acquisition of a memory skill. *Science, 208*(4448), 1181. doi:10.1126/science.7375930

Fonollosa, J., Neftci, E., & Rabinovich, M. (2015). Learning of Chunking Sequences in Cognition and Behavior. *PLoS computational biology, 11*(11), e1004592-e1004592. doi:10.1371/journal.pcbi.1004592

Gauthier, D. J. (2018). Reservoir Computing: Harnessing a Universal Dynamical System. Retrieved from https://sinews.siam.org/Details-Page/reservoir-computing-harnessing-a-universal-dynamical-system

Gee, J. P., & Grosjean, F. (1983). Performance structures: A psycholinguistic and linguistic appraisal. *Cognitive Psychology, 15*(4), 411-458. doi:https://doi.org/10.1016/0010-0285(83)90014-2

Gobet, F., Lane, P. C. R., Croker, S., Cheng, P. C. H., Jones, G., Oliver, I., & Pine, J. M. (2001). Chunking mechanisms in human learning. *Trends in Cognitive Sciences, 5*(6), 236-243. doi:10.1016/S1364-6613(00)01662-4

Graybiel, A. M. (1998). The Basal Ganglia and Chunking of Action Repertoires. *Neurobiology of Learning and Memory, 70*(1), 119-136. doi:https://doi.org/10.1006/nlme.1998.3843

Hinaut, X., & Dominey, P. F. (2013). Real-Time Parallel Processing of Grammatical Structure in the Fronto-Striatal System: A Recurrent Network Simulation Study Using Reservoir Computing. *PLOS ONE, 8*(2), e52946. doi:10.1371/journal.pone.0052946

Ho, V. M., Lee, J.-A., & Martin, K. C. (2011). The cell biology of synaptic plasticity. *Science (New York, N.Y.), 334*(6056), 623-628. doi:10.1126/science.1209236

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*.

Kelso, S. R., Ganong, A. H., & Brown, T. H. (1986). Hebbian synapses in hippocampus. *Proceedings of the National Academy of Sciences of the United States of America, 83*(14), 5326-5330. doi:10.1073/pnas.83.14.5326

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., . . . Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences, 114*(13), 3521. doi:10.1073/pnas.1611835114

Klibisz, A., & MacLennan, B. J. (2016). *A Primer on Reservoir Computing*.

Luck, S. J., & Vogel, E. K. (1997). The capacity of visual working memory for features and conjunctions. *Nature, 390*(6657), 279-281. doi:10.1038/36846

Lukoševičius, M., & Jaeger, H. (2009). Reservoir computing approaches to recurrent neural network training. *Computer Science Review, 3*(3), 127-149. doi:https://doi.org/10.1016/j.cosrev.2009.03.005

McCloskey, M., & Cohen, N. J. (1989). Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. In G. H. Bower (Ed.), *Psychology of Learning and Motivation* (Vol. 24, pp. 109-165): Academic Press.

Mehta, M. R. (2015). From synaptic plasticity to spatial maps and sequence learning. *Hippocampus, 25*(6), 756-762. doi:10.1002/hipo.22472

Miller, G. A. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review, 63*(2), 81-97. doi:10.1037/h0043158

Pammi, V. S. C., Miyapuram, K. P., Ahmed, Samejima, K., Bapi, R. S., & Doya, K. (2012). Changing the structure of complex visuo-motor sequences selectively activates the fronto-parietal network. *NeuroImage, 59*(2), 1180-1189. doi:https://doi.org/10.1016/j.neuroimage.2011.08.006

Rabinovich, M., Varona, P., Tristan, I., & Afraimovich, V. (2014). Chunking dynamics: heteroclinics in mind. *Frontiers in Computational Neuroscience, 8*(22). doi:10.3389/fncom.2014.00022

Rosenbaum, D. A., Kenny, S. B., & Derr, M. A. (1983). Hierarchical control of rapid movement sequences. *J Exp Psychol Hum Percept Perform, 9*(1), 86-102. doi:10.1037//0096-1523.9.1.86

Saffran, J. R., Aslin, R. N., & Newport, E. L. (1996). Statistical Learning by 8-Month-Old Infants. *Science, 274*(5294), 1926. doi:10.1126/science.274.5294.1926

Sakai, K., Kitaguchi, K., & Hikosaka, O. (2003). Chunking during human visuomotor sequence learning. *Experimental Brain Research, 152*(2), 229-242. doi:10.1007/s00221-003-1548-8

Salmon, D. P., & Butters, N. (1995). Neurobiology of skill and habit learning. *Current Opinion in Neurobiology, 5*(2), 184-190. doi:https://doi.org/10.1016/0959-4388(95)80025-5

Tanaka, G., Yamane, T., Héroux, J. B., Nakane, R., Kanazawa, N., Takeda, S., . . . Hirose, A. (2019). Recent advances in physical reservoir computing: A review. *Neural Networks, 115*, 100-123. doi:https://doi.org/10.1016/j.neunet.2019.03.005

Uddén, J., de Jesus Dias Martins, M., Zuidema, W., & Tecumseh Fitch, W. (2019). Hierarchical Structure in Sequence Processing: How to Measure It and Determine Its Neural Implementation. *Topics in Cognitive Science, n/a*(n/a). doi:10.1111/tops.12442

Vankayala, V. S. S., & Rao, N. D. (1993). Artificial neural networks and their applications to power systems—a bibliographical survey. *Electric Power Systems Research, 28*(1), 67-79. doi:https://doi.org/10.1016/0378-7796(93)90081-O

Verwey, W. B. (2001). Concatenating familiar movement sequences: the versatile cognitive processor. *Acta Psychologica, 106*(1), 69-95. doi:https://doi.org/10.1016/S0001-6918(00)00027-5

Verwey, W. B., & Eikelboom, T. (2003). Evidence for Lasting Sequence Segmentation in the Discrete Sequence-Production Task. *Journal of Motor Behavior, 35*(2), 171-181. doi:10.1080/00222890309602131

Wallisch, P., Lusignan, M., Benayoun, M., Baker, T. I., Dickey, A. S., & Hatsopoulos, N. G. (2009). Chapter 28 - Neural Networks Part I: Unsupervised Learning. In P. Wallisch, M. Lusignan, M. Benayoun, T. I. Baker, A. S. Dickey, & N. G. Hatsopoulos (Eds.), *Matlab for Neuroscientists* (pp. 307-317). London: Academic Press.

Widrow, B., Kim, Y., Park, D., & Perin, J. K. (2019). Chapter 1 - Nature's Learning Rule: The Hebbian-LMS Algorithm. In R. Kozma, C. Alippi, Y. Choe, & F. C. Morabito (Eds.), *Artificial Intelligence in the Age of Neural Networks and Brain Computing* (pp. 1-30): Academic Press.

Wymbs, Nicholas F., Bassett, Danielle S., Mucha, Peter J., Porter, Mason A., & Grafton, Scott T. (2012). Differential Recruitment of the Sensorimotor Putamen and Frontoparietal Cortex during Motor Chunking in Humans. *Neuron, 74*(5), 936-946. doi:10.1016/j.neuron.2012.03.038

Yadav, N., Yadav, A., & Kumar, M. (2015). An Introduction to Neural Network Methods for Differential Equations.

10.0 Appendix A

This network version has not been used for simulations in this project but was included for completeness as it was part of building the network structure used during the project. Version 1.1 was used for simulations for Experiment 1.

The reservoir computing network used in this thesis was implemented in Python 3.6. The Python library NumPy was used to build aspects of the network. This library gives support for large, multi-dimensional arrays and matrices, together with high level mathematical functions.

Version 1.0

```
"""
Created on 28-5-2020
Progam to simulate sequence memorization with a reservoir network
V1: first version
"""

import numpy as np
import random

# Adjusted logistic activation function for nodes. Input is incoming activation.
See perceptron file
# Input < 0 or = 0 results in 0 activation.
# Input > 0 results in activation, with maximum 1
# Maximum can be adapted by using key values variable top
def activation(Input, top=1):
if Input > 0:
r1 = np.exp(-Input)
r2 = top/(1 + r1)
r3 = 2*(r2 - 0.5)
else:
r3 = 0.0
return r3


# Function for generating connection matrix from firstNodes to secondNodes
# seed gives start randomization. sparsity gives chance of having a
# non-zero connection.
```

```
28. def connectionMatrix(seed, sparsity, firstNodes, secondNodes):
29. basisConnections = np.zeros(firstNodes*secondNodes).reshape(firstNodes, secondnodes)
30. random.seed(seed)
31. for i in range(firstNodes):      #For every node in firstNodes
32. for j in range(secondNodes):    #For every node in secondNodes
33. if random.random() > sparsity: # Decide if connection gets connenction value
34. basisConnections[i,j] = round(random.random(), 2)    #Decide connection value (=weight) randomly
35. return basisConnections
36.
37. # Function for generating reservoir matrix from resNodes to resNodes
38. # connections from node to itself are excluded
39. # seed gives start randomization. sparsity gives chance of having a non-zero connection.
40. def reservoirMatrix(seed, sparsity, resNodes):
41. basisReservoir = np.zeros(resNodes*resNodes).reshape(resNodes, resNodes)
42. random.seed(seed)
43. for i in range(resNodes):      #From every node
44. for j in range(resNodes):    #To every other node (but not itself)
45. if i != j:
46. if random.random() > sparsity:
47. basisReservoir[i,j] = round(random.random(), 2)    #Decide connection value (=weight) randomly
48. return basisReservoir
49.
50. # Initializations
51. inputNum = 5
52. inputSeed = 100
53. inputSparsity = 0.2
54. outputNum = 5
55. outputSeed = 200
56. outputSparsity = 0.2
57. resNum1 = 20
58. seedRes1 = 200
59. sparsityRes1 = 0.0
60.
61. # time parameter for development of activation in reservoir
62. timesteps = 10
63.
64.
65. #Connections from input to reservoir:
66. inputConnections = connectionMatrix(inputSeed, inputSparsity, inputNum, resNum1)
```

```
#Connections from reservoir to output:
outputConnections = connectionMatrix(outputSeed, outputSparsity, resNum1, outputNum)
#Connections in reservoir1
reservoir1 = reservoirMatrix(seedRes1, sparsityRes1, resNum1)

print('inputConnections')
print(inputConnections)
print('outputConnections')
print(outputConnections)
print('reservoir1')
print(reservoir1)

#Input nodes:
# For 5 elements: they stand for A B C D E
inputNodes = np.zeros(inputNum)

#Output nodes:
# For 5 elements: they stand for A B C D E
# Time is included as a parameter.
# We simulate the development of the activation of these nodes over time.
outputNodes = np.zeros(timeSteps*outputNum).reshape(timeSteps, outputNum)

# Nodes in reservoir1
# Time is included as a parameter.
# We simulate the development of the activation of these nodes over time.
resNodes1 = np.zeros(timeSteps*resNum1).reshape(timeSteps, resNum1)

#INPUT
#Give input to the reservoir: selecct one of the input nodes adn make it active (=1)
#Notice: python index starts with 0, and runs to 4 with 5 elements:
# Examle: input activation for element A: first node = 1. All others are 0.
# They can be activate them one by one (make the others zero again)
#This makes input sequence A B C D E
# Node for A:
inputNodes[0]= 1
inputNodes[1]= 0 #node for B
inputNodes[2]= 0 #node for C
inputNodes[3]= 0 #node for D
inputNodes[4]= 0 #node for E
```

```
106.
107.
108.
109. #Development of activation over time in reservoir and output
110. #This code calculates the new activation (in the next time step) in the reservoir nodes and the output nodes
111. #based on the inputs they get at the given time step
112. for i in range(timeSteps-1):
113. nextStep = i + 1
114. # activation of reservoir nodes
115. for q in range(resNum1):
116. resSum = 0.0
117. for k in range(inputNum):
118. resSum += inputConnections[k, q]*inputNodes[k]
119. for k in range(resNum1):
120. resSum += reservoir1[k, q]*resNodes1[i, k]
121. resNodes1[nextStep, q] =  activation(resSum)
122. # activation of output nodes
123. for q in range(outputNum):
124. outputSum = 0.0
125. for k in range(resNum1):
126. outputSum += outputConnections[k, q]*resNodes1[i, k]
127. outputNodes[nextStep, q] = activation(outputSum)
128.
129. """
130. In the program section below, the reservoir and output node activities over time are written in a file.
131. In Unix based systems (e.g. Mac), this file will be located in the directory (folder) of the program.
132. Otherwise, you might have to given the full path to a directory in the fiel name.
133. """
134.
135. outfile = open('Reservoir-Dynamics.txt', 'w')
136.
137. outfile.write('Evolution of the reservoir nodes over time:'  '\n')
138. for i in range(timeSteps):
139. for q in range(resNum1):
140. outfile.write('%7.2f' %resNodes1[i, q])
141. outfile.write('\n')
142. outfile.write('Evolution of the output nodes over time:'  '\n')
143. outfile.write('Node: A   B     C     D    E'  '\n')
144. for i in range(timeSteps):
```

```
for q in range(outputNum):
outfile.write('%7.2f' %outputNodes[i, q])
outfile.write('\n')
outfile.close()
```

Version 1.1

The reservoir computing network used in this thesis was implemented in Python 3.6. The Python library NumPy was used to build aspects of the network. This library gives support for large, multi-dimensional arrays and matrices, together with high level mathematical functions. The network was used for Experiment 1.

```
"""
Created on Mon Jun  1 21:14:35 2020

@author: lmjan
"""


import numpy as np
import random

# Adjusted logistic activation function for nodes. Input is incoming activation.
#See perceptron file
# Input < 0 or = 0 results in 0 activation.
# Input > 0 results in activation, with maximum 1
# Maximum can be adapted by using key values variable top
def activation(Input, top=1):
if Input > 0:
r1 = np.exp(-Input)
r2 = top/(1 + r1)
r3 = 2*(r2 - 0.5)
else:
r3 = 0.0
return r3


# Function for generating connection matrix from firstNodes to secondNodes
# seed gives start randomization. sparsity gives chance of having a non-zero connection.
```

```
28. def connectionMatrix(seed, sparsity, firstNodes, secondNodes):
29. basisConnections = np.zeros(firstNodes*secondNodes).reshape(firstNodes, secondNodes)
30. random.seed(seed)
31. for i in range(firstNodes):      #For every node in firstNodes
32. for j in range(secondNodes):    #For every node in secondNodes
33. if random.random() > sparsity: # Decide if connection gets connenction value
34. basisConnections[i,j] = round(random.random(), 2)    #Decide connection value (=weight) randomly
35. return basisConnections
36.
37. # Function for generating reservoir matrix from resNodes to resNodes
38. # connections from node to itself are excluded
39. # seed gives start randomization. sparsity gives chance of having a non-zero connection.
40. def reservoirMatrix(seed, sparsity, resNodes):
41. basisReservoir = np.zeros(resNodes*resNodes).reshape(resNodes, resNodes)
42. random.seed(seed)
43. for i in range(resNodes):      #From every node
44. for j in range(resNodes):    #To every other node (but not itself)
45. if i != j:
46. if random.random() > sparsity:
47. basisReservoir[i,j] = round(random.random(), 2)    #Decide connection value (=weight) randomly
48. return basisReservoir
49.
50. # Initializations
51. inputNum = 5
52. inputSeed = 100
53. inputSparsity = 0.3
54. outputNum = 5
55. outputSeed = 200
56. outputSparsity = 0.3
57. resNum1 = 10
58. seedRes1 = 200
59. sparsityRes1 = 0.3
60.
61. # time parameter for development of activation in reservoir
62. timeSteps = 10
63.
64.
65. #Connections from input to reservoir:
66. inputConnections = connectionMatrix(inputSeed, inputSparsity, inputNum, resNum1)
```

```
#Connections from reservoir to output:
outputConnections = connectionMatrix(outputSeed, outputSparsity, resNum1, outputNum)
#Connections in reservoir1
reservoir1 = reservoirMatrix(seedRes1, sparsityRes1, resNum1)

#print('inputConnections')
#print(inputConnections)
print('outputConnections')
print(outputConnections)
#print('reservoir1')
#print(reservoir1)

#Input nodes:
# For 5 elements: they stand for A B C D E
inputNodes = np.zeros(inputNum)


#Output nodes:
# For 5 elements: they stand for A B C D E
# Time is included as a parameter.
# We simulate the development of the activation of these nodes over time.
outputNodes = np.zeros(timeSteps*outputNum).reshape(timeSteps, outputNum)

# Nodes in reservoir1
# Time is included as a parameter.
# We simulate the development of the activation of these nodes over time.
resNodes1 = np.zeros(timeSteps*resNum1).reshape(timeSteps, resNum1)

#INPUT
#Give input to the reservoir: selecct one of the input nodes adn make it active (=1)
#Notice: python index starts with 0, and runs to 4 with 5 elements:
# Examle: input activation for element A: first node = 1. All others are 0.
# They can be activate them one by one (make the others zero again)
#This makes input sequence A B C D E

inputNodes[0]= 1 # Node for A
inputNodes[1]= 0 #node for B
inputNodes[2]= 0 #node for C
inputNodes[3]= 0 #node for D
```

```
inputNodes[4]= 0 #node for E

# Learning

bias_node = -1 # constant value
bias = 0.01 # weight of bias node
learning_rate = 0.1

# Actual output of Perceptron
outputConnectionsLearning = np.zeros(len(inputNodes))

# Desired output
outputDesired = np.zeros(len(inputNodes))
#Example
outputDesired[0]= 1 # desired output for input node for A
outputDesired[1]= 0 # desired output for input node for B
outputDesired[2]= 0 # desired output for input node for C
outputDesired[3]= 0 # desired output for input node for D
outputDesired[4]= 0 # desired output for input node for E

print('Desired output')
print(outputDesired)

repeats = 10

#Development of activation over time in reservoir and output
#This code calculates the new activation (in the next time step) in the reservoir nodes and the output nodes
#based on the inputs they get at the given time step

for r in range(repeats):
for i in range(timeSteps-1):
nextStep = i + 1
# activation of reservoir nodes
for q in range(resNum1):
resSum = 0.0
for k in range(inputNum):
resSum += inputConnections[k, q]*inputNodes[k]
for k in range(resNum1):
resSum += reservoir1[k, q]*resNodes1[i, k]
```

```
145. resNodes1[nextStep, q] =  activation(resSum)
146. # activation of output nodes
147. for q in range(outputNum):
148. outputSum = 0.0
149. for k in range(resNum1):
150. outputSum += outputConnections[k, q]*resNodes1[i, k]
151. outputNodes[nextStep, q] = activation(outputSum)
152.
153. # Learning
154. for q in range(outputNum):
155. for k in range(resNum1):
156. outputConnections[k, q] += (outputDesired[q] - outputNodes[-1, q])*learning_rate
157. if outputConnections[k, q] < 0:
158. outputConnections[k, q] = 0
159.
160. # Learning
161. for q in range(outputNum):
162. for k in range(resNum1):
163. outputConnections[k, q] += (outputDesired[q] - outputNodes[-1, q])*learning_rate
164. if outputConnections[k, q] < 0:
165. outputConnections[k, q] = 0
166.
167.
168. print('outputConnections')
169. print(outputConnections)
170.
171.
172.
173.
174. """
175. In the program section below, the reservoir and output node activities over time are written in a file.
176. In Unix based systems (e.g. Mac), this file will be located in the directory (folder) of the program.
177. Otherwise, you might have to given the full path to a directory in the fiel name.
178. """
179.
180. outfile = open('Reservoir-Dynamics.C.5.txt', 'w')
181.
182. ##outfile.write('Evolution of the reservoir nodes over time:'  '\n')
183. ##for i in range(timeSteps):
```

```
##     for q in range(resNum1):
##         outfile.write('%7.2f' %resNodes1[i, q])
##     outfile.write('\n')
outfile.write('Evolution of the output nodes over time:'  '\n')
outfile.write('Node: A   B     C     D    E'  '\n')
for i in range(timeSteps):
for q in range(outputNum):
outfile.write('%7.2f' %outputNodes[i, q])
outfile.write('\n')
outfile.close(
```

10.1 Appendix B

The reservoir computing network used in this thesis was implemented in Python 3.6. The Python library NumPy was used to build aspects of the network. This library gives support for large, multi-dimensional arrays and matrices, together with high level mathematical functions. The network was used for Experiment 2.

Version 2.0

```
Created June 2020
Program for presentation, learning and reactivation of sequence of items
Start sequence with 5 items: A, B, C, D, E
Aim: learn sequence in reservoir so that whole sequence is reactivated by presening A

Reversing role connections::
Connections from input to reservoir and from reservoir to output are fixed.
Learning in reservoir: genuine learning in the time domain.
Learning: Hebbian

"""

import numpy as np
import random

## Adjusted logistic activation function for nodes. Input is incoming activation. See perceptron file
## Input < 0 or = 0 results in 0 activation.
```

```
18. ## Input > 0 results in activation, with maixumum 1
19. ## Maximum can be adapted by using key values variable top
20. def activation(Input, top=1):
21. if Input > 0:
22. r1 = np.exp(-Input)
23. r2 = top/(1 + r1)
24. r3 = 2*(r2 - 0.5)
25. else:
26. r3 = 0.0
27. return r3
28.
29.
30. # Function for generating reservoir matrix from resNodes to resNodes
31. # connections from node to itself are excluded
32. # seed gives start randomization. sparsity gives chance of having a non-zero connection.
33. def reservoirMatrix(seed, sparsity, resNodes):
34. basisReservoir = np.zeros(resNodes*resNodes).reshape(resNodes, resNodes)
35. random.seed(seed)
36. for i in range(resNodes):      # From every node
37. for j in range(resNodes):    # To every other node (but not itself)
38. basisReservoir[i,j] = -1   # means: connection does not exist
39. if i != j:
40. if random.random() > sparsity:
41. basisReservoir[i,j] = 0    # connection exits (but no weight yet).
42. return basisReservoir
43.
44. # Initializations
45. inputNum = 5
46. outputNum = inputNum
47. resNum1 = 10
48. resSeed1 = 10
49. resSparsity1 = 0.0   # 0: all nodes are interconnected (except node to itself)
50.
51. # use fixed connection weight
52. connectionWeight = 1
53. # learning parameter for Hebbian Learning
54. hebb= 0.2
55.
56. # time parameter for presentation, learning and reactivation sequence
```

```
# here: same as number of items in sequence
time = 5

# The section below randomly selects nodes from the reservoir
# They are used to represent the items of the sequence in the reservoir
# Here: One specific (selective, unique) reservoir node for each item in sequence
# They are connected to the input and output nodes with fixed connections

# make list of index numbers of reservoir nodes
resList = []
for i in range(resNum1):
resList.append(i)

# randomly shuffle list of index numbers of reservoir nodes
seedShuffle = 5
random.Random(seedShuffle).shuffle(resList)
print('shuffled resList')
print(resList)

# The first 5 index numbers in the shuffled index list are the reservoir nodes
# that represent the items in the reservoir (in the order A, B, C, D, E)


# Initialization of 2D arrays for nodes. Variables: (time, index)
inputNodes = np.zeros(time*inputNum).reshape(time, inputNum)
outputNodes = np.zeros(time*outputNum).reshape(time, outputNum)
resNodes1 = np.zeros(time*resNum1).reshape(time, resNum1)

# Fixed connections:
# From input nodes to reservoir nodes
inputConnections = np.zeros(resNum1*inputNum).reshape(resNum1,inputNum)
# From reservoir nodes to output nodes
outputConnections = np.zeros(resNum1*outputNum).reshape(outputNum, resNum1)
# Filling in fixed connections:
for i in range(inputNum):
inputConnections[resList[i], i] = connectionWeight
outputConnections[i, resList[i]] = connectionWeight

#Connections in reservoir1
```

```
resConnections1 = reservoirMatrix(resSeed1, resSparsity1, resNum1)


# Presentation and learning of sequence
repeat = 30    # number of repetitions of sequence presentation and learning

for r in range(repeat):
# sequential activation and learning of sequence
for q in range(time):
# sequential activation input nodes
for i in range(inputNum):
inputNodes[q, i ] = 0  # set input nodes 0 before activating item in sequence
for s in range(time):   # select the input node for the active item in sequence
if q == s and i == s:
inputNodes[q, i ] = 1

# sequential activation of the reservoir nodes by input nodes
for i in range(resNum1):
resNodes1[q, i] = 0  # activation of res nodes is 0 at beginning of each time step
# input from input Nodes
inputSum = 0.0
for k in range(inputNum):
inputSum += inputConnections[i, k]*inputNodes[q, k]
# input from res nodes active in previous time step
reservoirSum = 0.0
if q > 0:
for k in range(resNum1):
if resConnections1[i, k] != -1:   # connection exits
reservoirSum += resConnections1[i, k]*resNodes1[q-1, k]
resNodes1[q, i] = round(activation(inputSum + reservoirSum), 2)

# sequential activation of the output nodes by res nodes
for i in range(outputNum):
outputNodes[q, i] = 0    #activation of output nodes is 0 at beginning of each time step
outputSum = 0.0
for k in range(resNum1):
outputSum += outputConnections[i, k]*resNodes1[q, k]
outputNodes[q, i] = round(activation(outputSum), 2)

```

```
# sequential learning: from presynaptic node to post synaptic node in reservoir
if q > 0:
for i in range(resNum1):
for k in range(resNum1):
if resConnections1[i, k] != -1:   # connection exits
# Hebbian Learning:
resConnections1[i, k] += round(resNodes1[q, i]*resNodes1[q-1, k]*hebb, 2)


print('inputConnections')
print(inputConnections)
print('outputConnections')
print(outputConnections)
print('resConnections1')
print(resConnections1)

print(' input nodes')
print(inputNodes)
print('resNodes1')
print(resNodes1)
print('outputNodes')
print(outputNodes)


# Rectivation of sequence by first item in sequence:
for q in range(time):
# start activation of the input nodes
for i in range(inputNum):
inputNodes[q, i ] = 0
if q == 0 and i ==0:    # activate input of first item (A) at time = 0 (start)
inputNodes[q, i ] = 1

# sequential activation of the res nodes by input nodes
for i in range(resNum1):
resNodes1[q, i] = 0  # activation of res nodes is 0 at beginning of each time step
# input from input Nodes
inputSum = 0.0
for k in range(inputNum):
inputSum += inputConnections[i, k]*inputNodes[q, k]
```

```
# input from res nodes active in previous time step
reservoirSum = 0.0
if q > 0:
for k in range(resNum1):
if resConnections1[i, k] != -1:   # connection exits
reservoirSum += resConnections1[i, k]*resNodes1[q-1, k]
resNodes1[q, i] = round(activation(inputSum + reservoirSum), 2)

# sequential activation of the output nodes by res nodes
for i in range(outputNum):
outputNodes[q, i] = 0    #activation of output nodes is 0 at beginning of each time step
outputSum = 0.0
for k in range(resNum1):
outputSum += outputConnections[i, k]*resNodes1[q, k]
outputNodes[q, i] = round(activation(outputSum), 2)


print('reactivation input nodes')
print(inputNodes)

print('reactivation resNodes1')
print(resNodes1)

print('reactivation outputNodes')
print(outputNodes)
```

10.2 Appendix C

The reservoir computing network used in this thesis was implemented in Python 3.6. The Python library NumPy was used to build aspects of the network. This library gives support for large, multi-dimensional arrays and matrices, together with high level mathematical functions. The network was used for Experiment 3 & 4.

Version 3.0

```
"""
Created June 2020
Program for presentation, learning and reactivation of sequence of items
```

```
4.  Start sequence with 5 items: A, B, C, D, E
5.  Aim: learn sequence in reservoir so that whole sequence is reactivated by presening A
6.
7.  Reversing role connections::
8.  Connections from input to reservoir and from reservoir to output are fixed.
9.  Learning in reservoir: genuine learning in the time domain.
10. Learning: Hebbian
11.
12. NEW 30-6-20:
13. cluster: select a cluster of reservoir nodes to represent items (A, B, etc) in reservoir
14. Needed for multiple occurrences of same item in sequence, as in e.g.,  A, B, A, C, D, E
15. Size of cluster is the same for each item.
16. So: cluster size * number of items < reservoir size
17. Here: clusterNum*inputNum < resNum1
18. """
19.
20. import numpy as np
21. import random
22.
23. ## Adjusted logistic activation function for nodes. Input is incoming activation. See perceptron file
24. ## Input < 0 or = 0 results in 0 activation.
25. ## Input > 0 results in activation, with maixumum 1
26. ## Maximum can be adapted by using key values variable top
27. def activation(Input, top=1):
28. if Input > 0:
29. r1 = np.exp(-Input)
30. r2 = top/(1 + r1)
31. r3 = 2*(r2 - 0.5)
32. else:
33. r3 = 0.0
34. return r3
35.
36.
37. # Function for generating reservoir matrix from resNodes to resNodes
38. # connections from node to itself are excluded
39. # seed gives start randomization. sparsity gives chance of having a non-zero connection.
40. def reservoirMatrix(seed, sparsity, clusters):
41. #cluster for each item, i.e. array containing one or more nodes that represent one item
42. basisReservoir = np.zeros(len(clusters) * len(clusters) * len(clusters[0])).reshape(
```

```
43. len(clusters), len(clusters), len(clusters[0]))
44. random.seed(seed)
45. for i in range(len(basisReservoir)):              # From every cluster
46. for j in range(len(basisReservoir[i])):       # To every other cluster (but not itself)
47. for k in range(len(basisReservoir[i, j])):  # For each node in the cluster
48. basisReservoir[i,j,k] = -1            # means: connection does not exist
49. if i != j:
50. if random.random() > sparsity:
51. basisReservoir[i,j,k] = 0      # connection exits (but no weight yet).
52. return basisReservoir
53.
54. # Initializations
55. inputNum = 10
56. outputNum = inputNum
57. clusterNum = 8
58. resNum1 = 100
59. resSeed1 = 10
60. resSparsity1 = 0.0  # 0: all nodes are interconnected (except node to itself)
61.
62. # use fixed connection weight
63. connectionWeight = 1
64. # learning parameter for Hebbian Learning
65. hebb= 0.2
66.
67. # time parameter for presentation, learning and reactivation sequence
68. # here: same as number of items in sequence
69. time = 5
70.
71. # The section below randomly selects nodes from the reservoir
72. # They are used to represent the items of the sequence in the reservoir
73. # Here: One specific (selective, unique) reservoir node for each item in sequence
74. # They are connected to the input and output nodes with fixed connections
75.
76. # make list of index numbers of reservoir nodes
77. resList = []
78. for i in range(resNum1):
79. resList.append(i)
80.
81. # randomly shuffle list of index numbers of reservoir nodes
```

```
seedShuffle = 5
random.Random(seedShuffle).shuffle(resList)
print('shuffled resList')
print(resList)

# The first clusterNum*inputNum index numbers in the shuffled index list are the reservoir nodes
# that represent the items in the reservoir (in the order A, B, C, D, E)

# making array to store the index numbers that represent the items in the reservoir
itemsRes = []
for i in range(inputNum):
itemsRes.append([ ])
for k in range(clusterNum):
itemsRes[i].append(k)

# filling array to store the index numbers that represent the items in the reservoir
counter = 0
for i in range(inputNum):
for k in range(clusterNum):
itemsRes[i][k] = resList[counter]
counter += 1

print('itemsRes')
print(itemsRes)

# Initialization of 2D arrays for nodes. Variables: (time, index)
inputNodes = np.zeros(time*inputNum).reshape(time, inputNum)
outputNodes = np.zeros(time*outputNum).reshape(time, outputNum)
resNodes1 = np.zeros(time*resNum1).reshape(time, resNum1)

# Fixed connections:
# From input nodes to reservoir nodes
inputConnections = np.zeros(resNum1 * inputNum).reshape(resNum1,inputNum)
# From reservoir nodes to output nodes
outputConnections = np.zeros(outputNum * resNum1).reshape(outputNum, resNum1)
# Filling in fixed connections:
# Adapted fro clusters in reservoir
for i in range(inputNum):
for k in range(clusterNum):
```

```
121. inputConnections[itemsRes[i][k], i] = connectionWeight
122. outputConnections[i, itemsRes[i][k] ] = connectionWeight
123.
124. #Connections in reservoir1
125. resConnections1 = reservoirMatrix(resSeed1, resSparsity1, itemsRes)
126.
127. # Presentation and learning of sequence
128. repeat = 50    # number of repetitions of sequence presentation and learning1
129.
130. inputNodes = np.array([[ 1, 0, 0, 0, 0 ],
131. [ 0, 1, 0, 0, 0 ],
132. [ 1, 0, 0, 0, 0 ],
133. [ 0, 0, 0, 1, 0 ],
134. [ 0, 0, 0, 0, 1 ],
135. [ 0, 1, 0, 0, 0 ],
136. [ 1, 0, 0, 0, 0 ],
137. [ 0, 0, 0, 1, 0 ],
138. [ 0, 0, 0, 0, 1 ],
139. [ 1, 0, 0, 0, 0 ]])
140.
141. for r in range(repeat):
142. # sequential activation and learning of sequence
143. for t in range(len(resNodes1)):
144. for node in range(len(resNodes1[t])):
145. resNodes1[t, node] = 0
146.
147. nodecounter = [0] * inputNum
148. for q in range(time):
149. for postcluster in range(len(itemsRes)):
150. postsyn = itemsRes[postcluster][nodecounter[postcluster]]
151. inputSum = inputConnections[postsyn, postcluster] * inputNodes[q, postcluster]
152.
153. reservoirSum = 0.0
154. for precluster in range(len(itemsRes)):
155. for prenode in range(len(itemsRes[precluster])):
156. if resConnections1[precluster, postcluster, prenode] != -1:
157. reservoirSum += (resConnections1[precluster, postcluster, prenode] *
158. resNodes1[q-1, itemsRes[precluster][prenode]])
159.
```

```
160. resNodes1[q, postsyn] += round(activation(inputSum + reservoirSum), 2)
161. # if the currentnode got any activation this q-loop,
162. # it cannot get any activation anymore in next q-loops
163. # so set the counter for this cluster to +1
164. # BREAKS IF THE CURRENT CLUSTER APPEARS MORE OFTEN IN THE INPUTLIST
165. # THAN THERE ARE NODES IN EACH CLUSTER!!
166. if resNodes1[q, postsyn] > 0 and nodecounter[postcluster] < (clusterNum-1):
167. nodecounter[postcluster] += 1
168.
169. # Hebbian learning to adjust reservoir connections
170. if q > 0:
171. postsyn = None
172. postsynactivity = 0.0
173. presyn = None
174. presynactivity = 0.0
175. for cluster in range(len(itemsRes)):
176. for node in range(len(itemsRes[postcluster])):
177. if resNodes1[q, itemsRes[cluster][node]] > 0:
178. postsyn = [cluster, node]
179. postsynactivity = resNodes1[q, itemsRes[cluster][node]]
180. if resNodes1[q-1, itemsRes[cluster][node]] > 0:
181. presyn = [cluster, node]
182. presynactivity = resNodes1[q-1, itemsRes[cluster][node]]
183.
184. connstr = round(presynactivity * postsynactivity * hebb, 2)
185. resConnections1[presyn[0], postsyn[0], presyn[1]] += connstr
186.
187. # sequential activation of the output nodes by res nodes
188. for i in range(outputNum):
189. outputNodes[q, i] = 0    # activation of output nodes is 0 at beginning of each time step
190. outputSum = 0.0
191. for k in range(resNum1):
192. outputSum += outputConnections[i, k]*resNodes1[q, k]
193. outputNodes[q, i] = round(activation(outputSum), 2)
194.
195. print('inputConnections')
196. print(inputConnections)
197. # print('outputConnections')
198. # print(outputConnections)
```

```
print('resConnections1')
print(resConnections1)

print(' input nodes')
print(inputNodes)
print('resNodes1')
print(resNodes1)
print('outputNodes')
print(outputNodes)


inputNodes = np.array([[ 1, 0, 0, 0, 0 ],
[ 0, 1, 0, 0, 0 ],
[ 1, 0, 0, 0, 0 ],
[ 0, 0, 0, 1, 0 ],
[ 0, 0, 0, 0, 1 ],
[ 0, 1, 0, 0, 0 ],
[ 1, 0, 0, 0, 0 ],
[ 0, 0, 0, 1, 0 ],
[ 0, 0, 0, 0, 1 ],
[ 1, 0, 0, 0, 0 ]])

for t in range(len(resNodes1)):
for node in range(len(resNodes1[t])):
resNodes1[t, node] = 0

nodecounter = [0] * inputNum
for q in range(time):
for postcluster in range(len(itemsRes)):
postsyn = itemsRes[postcluster][nodecounter[postcluster]]
inputSum = inputConnections[postsyn, postcluster] * inputNodes[q, postcluster]

reservoirSum = 0.0
for precluster in range(len(itemsRes)):
for prenode in range(len(itemsRes[precluster])):
if resConnections1[precluster, postcluster, prenode] != -1:
reservoirSum += (resConnections1[precluster, postcluster, prenode] *
resNodes1[q-1, itemsRes[precluster][prenode]])

```

```
238. resNodes1[q, postsyn] += round(activation(inputSum + reservoirSum), 2)
239. # if the currentnode got any activation this q-loop,
240. # it cannot get any activation anymore in next q-loops
241. # so set the counter for this cluster to +1
242. # BREAKS IF THE CURRENT CLUSTER APPEARS MORE OFTEN IN THE INPUTLIST
243. # THAN THERE ARE NODES IN EACH CLUSTER!!
244. if resNodes1[q, postsyn] > 0 and nodecounter[postcluster] < (clusterNum-1):
245. nodecounter[postcluster] += 1
246.
247. # sequential activation of the output nodes by res nodes
248. for i in range(outputNum):
249. outputNodes[q, i] = 0    # activation of output nodes is 0 at beginning of each time step
250. outputSum = 0.0
251. for k in range(resNum1):
252. outputSum += outputConnections[i, k]*resNodes1[q, k]
253. outputNodes[q, i] = round(activation(outputSum), 2)
254.
255.
256.
257. print('reactivation input nodes')
258. print(inputNodes)
259.
260. print('reactivation resNodes1')
261. print(resNodes1)
262.
263. print('reactivation outputNodes')
264. print(outputNodes)
```

10.3 Appendix D

The reservoir computing network used in this thesis was implemented in Python 3.6. The Python library NumPy was used to build aspects of the network. This library gives support for large, multi-dimensional arrays and matrices, together with high level mathematical functions. The network was used for Experiment 5.

Version 4.0

```
"""
Created August 2020
Program to learn set of  sequences in reservoir (R).
Based on program ReservoirNewCluster.py (with modifications)

Sequences of 5 items: A, B, C, D, E
Using number coding:
A=0, B=1, C=2, D=3, E=4
(can be more sophisticated: that is for another project)

Two types of clusters of R nodes:
start cluster: one R node for each item. So, size is equal to number of items
item clusters: one cluster for each item
Size of item clusters is variable, but is the same for each item.
So: cluster size * number of items + startcluster < reservoir size
Here: clusterNum*inputNum + inputNum < resNum1
"""

import numpy as np
import random

## Adjusted logistic activation function for nodes. Input is incoming activation. See perceptron file
## Input < 0 or = 0 results in 0 activation.
## Input > 0 results in activation, with maximum 1
def activation(Input, top=1):
if Input > 0:
r1 = np.exp(-Input)
r2 = top/(1 + r1)
r3 = 2*(r2 - 0.5)
else:
r3 = 0.0
return r3

# Function for generating reservoir matrix from resNodes to resNodes
# connections from node to itself are excluded
# seed gives start randomization. sparsity gives chance of having a non-zero connection.
def reservoirMatrix(seed, sparsity, resNodes):
```

```
basisReservoir = np.zeros(resNodes*resNodes).reshape(resNodes, resNodes)
random.seed(seed)
for i in range(resNodes):      # From every node
for j in range(resNodes):    # To every other node (but not itself)
basisReservoir[i,j] = -1   # means: connection does not exist
if i != j:
if random.random() > sparsity:
basisReservoir[i,j] = 0    # connection exits (but no weight yet).
return basisReservoir

# Initializations
inputNum = 5          # number of items in individual sequence
outputNum = inputNum
clusterNum = 20        # number of R nodes in cluster for each item in sequence (not for start cluster)
resNum1 = 150         # size of the reservoir (R)
resSeed1 = 10
resSparsity1 = 0.7    # 0: all nodes are interconnected (except node to itself)
numSeq = 5           # number of sequences that are learned

# use fixed connection weight
connectionWeight = 1

# learning parameter for Hebbian Learning
hebb= 0.2
maxW = 5.0  # maximum value of the learned weights

# input activation representing presentation of items
inputActivation = 1

# time parameter for presentation, learning and reactivation sequence
# here: same as number of items in sequence
time = 5

# number of repetitions of sequence presentation and learning
repeat = 30


# The section below randomly selects nodes from the reservoir
# They are used to represent the items of the sequence in the reservoir
```

```
77. # by filling in the start and item clusters
78. # They are connected to output nodes with fixed connections
79.
80. # make list of index numbers of R nodes
81. resList = []
82. for i in range(resNum1):
83. resList.append(i)       # first item in list: 0
84.
85. print('resList')
86. print(resList)
87.
88. # randomly shufflet the list of index numbers of R nodes
89. seedShuffle = 5
90. random.Random(seedShuffle).shuffle(resList)
91. print('shuffled resList')
92. print(resList)
93.
94. # The first 5 R nodes of the shufled R list are the nodes for the start items of each sequence.
95. # with the first = A, the second = B etc.
96. startNodes = []
97. for i in range(inputNum):
98. startNodes.append(resList[i])
99.
100. print('startNodes')
101. print(startNodes)
102.
103. # making array to store the index numbers that represent the item clusters in R
104. itemsRes = []
105. for i in range(inputNum):
106. itemsRes.append([ ])
107. for k in range(clusterNum):
108. itemsRes[i].append(k)
109.
110. # filling array to store the index numbers that represent the item clusters in R
111. # now itemRes starts at resList[5], because of startNodes
112. # cluster_counter counts number of R nodes in an item cluster
113. # It starts at 5 because first 5 R nodes from shufled R list are in the start cluster
114. cluster_counter = 5
115. for i in range(inputNum):
```

```
for k in range(clusterNum):
itemsRes[i][k] = resList[cluster_counter]
cluster_counter += 1

print('itemsRes')
print(itemsRes)

# Initialization of 2D arrays for nodes. Variables: (time, index)
outputNodes = np.zeros(time*outputNum).reshape(time, outputNum)
resNodes1 = np.zeros(time*resNum1).reshape(time, resNum1)
start_resNodes1 = np.zeros(time*resNum1).reshape(time, resNum1)  # used for input to R

# Fixed connections:
# From reservoir nodes to output nodes
outputConnections = np.zeros(resNum1*outputNum).reshape(outputNum, resNum1)
# Filling in fixed connections:
# Adapted for clusters in reservoir
for i in range(inputNum):
outputConnections[i, startNodes[i]] = connectionWeight
for k in range(clusterNum):
outputConnections[i, itemsRes[i][k] ] = connectionWeight

#Connections in reservoir1
resConnections1 = reservoirMatrix(resSeed1, resSparsity1, resNum1)

# Making the item sequences. First element is start item
itemSeqs = np.array([[ 0, 1, 2, 0, 4],
[ 1, 2, 4, 0, 1],
[ 0, 1, 2, 1, 4],
[ 3, 0, 4, 2, 1],
[ 4, 0, 4, 2, 1]])

print('itemSeqs')
print(itemSeqs)

# Array to count number of R nodes in item clusters that are (have been) activated
# Used to activate next R node in an item cluster
count = np.zeros(inputNum)

```

```
155. # open a file to print results. Here: resConnections1 for each learned sequence
156. outfile = open('Res-connections.dat', 'w')
157.
158. # Start main loop for presenting and simulating (learning) set of sequences
159. for seq in range(numSeq):
160.
161. # First: clean resNodes1 and start_resNodes1 for new sequence
162. # note: count is NOT cleaned: used to remember number of cluster nodes that have been activated
163. for q in range(time):
164. for k in range(resNum1):
165. resNodes1[q, k] = 0
166. start_resNodes1[q, k] = 0
167.
168. # Then: initilalze start_resNodes1 for sequence
169. for q in range(time):
170. if q == 0:
171. start_resNodes1[q, startNodes[itemSeqs[seq, q]]] =  round(activation(inputActivation), 2)
172. if q > 0:
173. for k in range(inputNum):
174. if itemSeqs[seq, q] == k:
175. clusterIndex = int(count[k])     # count is a numpy array: cannot be used as index directly
176. start_resNodes1[q, itemsRes[k][clusterIndex]] = round(activation(inputActivation), 2)
177. count[k] += 1
178.
179. # print results:
180. print('count')
181. print(count)
182. print('start_resNodes1')
183. print(start_resNodes1)
184.
185. # Presentation and learning of sequence. Here: loop over repeat
186. for r in range(repeat):
187. # sequential activation and learning of sequence
188. for q in range(time):
189. # input from res nodes active in previous time step
190. for i in range(resNum1):
191. reservoirSum = 0.0
192. if q > 0:
193. for k in range(resNum1):
```

```
194. if resConnections1[i, k] != -1:   # connection exits
195. reservoirSum += resConnections1[i, k]*resNodes1[q-1, k]
196. resNodes1[q, i] = round(activation(reservoirSum), 2) + start_resNodes1[q, i]
197.
198. # sequential activation of the output nodes by res nodes
199. for i in range(outputNum):
200. outputNodes[q, i] = 0    #activation of output nodes is 0 at beginning of each time step
201. outputSum = 0.0
202. for k in range(resNum1):
203. outputSum += outputConnections[i, k]*resNodes1[q, k]
204. outputNodes[q, i] = round(activation(outputSum), 2)
205.
206. # sequential learning: from presynaptic node to post synaptic node in reservoir
207. if q > 0:
208. for i in range(resNum1):
209. for k in range(resNum1):
210. if resConnections1[i, k] != -1:   # connection exits
211. # Hebbian Learning:
212. resConnections1[i, k] += round(resNodes1[q, i]*resNodes1[q-1, k]*hebb, 2)
213. if resConnections1[i, k] > maxW:
214. resConnections1[i, k] = maxW
215.
216. # print results
217. print('sequence number  ', seq)
218. print('outputConnections')
219. print(outputConnections)
220. print('activation resNodes1')
221. print(resNodes1)
222.
223. ##    print('resConnections1')
224. ##    for i in range(resNum1):
225. ##        print(resConnections1[i])
226.
227. # writing resConnections1 to data file
228. outfile.write('sequence number: ')
229. outfile.write(' %2d  ' %seq)
230. outfile.write('\n')
231. for i in range(resNum1):
232. for k in range(resNum1):
```

```
233. outfile.write('%7.2f' %resConnections1[i,k])
234. outfile.write('\n')
235.
236. outfile.close()
237.
238. # Reactivation
239. # Below: row is time, column is node (identitiy)
240.
241. print('start of reactivation')
242.
243. # Reactivation of sequence by first item in sequence:
244. for seq in range(numSeq):
245.
246. # clean resNodes1 and start_resNodes1 for new sequence
247. for q in range(time):
248. for k in range(resNum1):
249. resNodes1[q, k] = 0
250. start_resNodes1[q, k] = 0
251.
252. # initilalze for sequence
253. for q in range(time):
254. if q == 0:
255. start_resNodes1[q, startNodes[itemSeqs[seq, q]]] =  round(activation(inputActivation), 2)
256.
257. for q in range(time):
258. # input from res nodes active in previous time step
259. for i in range(resNum1):
260. reservoirSum = 0.0
261. if q > 0:
262. for k in range(resNum1):
263. if resConnections1[i, k] != -1:   # connection exits
264. reservoirSum += resConnections1[i, k]*resNodes1[q-1, k]
265. resNodes1[q, i] = round(activation(reservoirSum), 2) + start_resNodes1[q, i]
266.
267. # sequential activation of the output nodes by res nodes
268. for i in range(outputNum):
269. outputNodes[q, i] = 0    # activation of output nodes is 0 at beginning of each time step
270. outputSum = 0.0
271. for k in range(resNum1):
```

```
272. outputSum += outputConnections[i, k]*resNodes1[q, k]
273. outputNodes[q, i] = round(activation(outputSum), 2)
274.
275. print('sequence number  ', seq)
276. print('reactivation resNodes1')
277. print(resNodes1)
278. print('reactivation outputNodes')
279. print(outp)
```