MASTER THESIS

Applying Reservoir Computing to Semantic Networks

Simulating the use of a boron dopant cell as a reservoir

Robbin Koopman
S1716018

Master Human Factors & Engineering Psychology
Faculty of Behavioural, Management and Social Sciences (BMS)

EXAMINATION COMMITTEE
First supervisor: Prof. Dr. F. van der Velde
Second supervisor: Prof. Dr. Ir. B.J. Geurts

DATE
25-08-2021

UNIVERSITY OF TWENTE.

Abstract

Semantic analysis, a process within Natural Language Analysis (NLA), requires working with a large corpus of information called a semantic network. Modern-day computers are ill-equipped for dealing with this amount of data in a flexible manner. Therefore, other solutions are being researched in the domain of neuromorphic computing. Chen et al. (2020) have developed such a solution in the form of a boron dopant cell, that is capable of solving logic gates. The goal of the current research is to find out whether such a system could be utilized to run a small semantic network. This is done by simulating the boron dopant cell using reservoir computing and applying it to aspects of the Parallel Distributed Processing (PDP) model (McClelland and Rogers, 2003). In the first four experiments, different structures are tested using one or more reservoirs and simple control nodes for learning. The results show moderate to high success for linear gates, but a low success rate for non-linear gates due to unequal contribution of intermediate gates. In the fifth experiment, saturation cells are added to attempt to counter this problem. Lastly, the PDP model is implemented by using one reservoir for each attribute and solving the AND gate for each attribute. When using the same seed configuration for each reservoir, there is a moderate success rate, but using unique reservoirs does not result in successful iterations.

Table of contents

1. Introduction

Natural Language Processing (NLP) is a type of Artificial Intelligence that deals with the processing of human language by a computer system. As a subdomain of linguistics, NLP concerns all the computational methods of analyzing and representing 'normal' human language in a way similar to humans. Applications of NLP include virtual chat agents on websites, spam filters for e-mail, and translation of text (Sharma, 2020).

A distinction is often made within NLP between language analysis and language generation (Chowdhary, 2020; Liddy, 2001). Natural Language Analysis (NLA) is about processing or translating input into a meaningful representation. NLA processes entities from small to big; starting with words, then sentences, then the text as a whole. The semantic meaning of a sentence or complete text may influence the interpretation of specific words, so NLA is typically not a linear process (Liddy, 2001).

Natural Language Generation (NLG) concerns the production of language by a computer system. This consists out of three main tasks; 1). Determine content and plan how to structure it; 2). Decide how to split information into sentences and paragraphs; and 3). Generate sentences that are grammatically correct (Reiter & Dale, 1997). These steps require a lot of the same processes also required for NLA, with the additional requirement of planning what to communicate, and in what manner (Liddy, 2001). The scope of this thesis is therefore limited to NLA, as there is still much to gain in that area, which would simultaneously benefit NLG.

There are different ways of structuring Natural Language Analysis. Chowdhary (2020) proposes a branch structure, in which Language Analysis is broken down into Sentence Analysis and Discourse Analysis. Sentence Analysis refers to the processing per sentence, whereas Discourse Analysis goes beyond one sentence and takes multiple sentences or even a full text into consideration. Sentence Analysis, in turn, branches into Syntax Analysis and Semantic Analysis. Syntax Analysis is about determining the structure of the sentence, or to simplify it for the next steps of analysis. Semantic Analysis aims at interpreting the subject matter of a sentence (Chowdhary, 2020).

Another way of structuring NLA is by the different levels of linguistic analysis (Hausser, 2001; Khurana et al., 2017; Liddy, 2001). These levels concern an increasing part of the overall text, but are not necessarily followed in a sequential order, as higher order processing may influence lower levels of analysis (Liddy, 2001). The levels within this model

are phonology, morphology, lexical, syntactic, semantic, and pragmatic (Hausser, 2014). Liddy (2001) names an additional step between semantic and pragmatic processing, namely discourse analysis.

Many steps in this process have been implemented already in some practical applications. Especially the lower-order processes (i.e., phonology, morphology and parts of the lexical analysis) are mostly rule-based and therefore relatively easy for computer systems to simulate (Liddy, 2001). The higher-order processes deal with more complexity, as there are no simple yes-or-no rules to determine the correct interpretation of a certain word or sentence as a whole within a specific context. One of the main difficulties lies in the lack of computational power with modern-day systems, as these processes typically require working with a large amount of data.

In the following section, the steps in NLA are outlined in more detail to put the level of semantic analysis in more perspective. However, the scope of this thesis is limited to issues of computing power related to the semantic analysis part of NLA, as the lower-order rule-based processes can be achieved with relatively simple algorithms. The last steps (i.e., discourse and pragmatic analysis) depend on a functional semantic analysis as discussed here, because it is impossible to determine the meaning of an entire text when the definitions of individual words are unknown or uncertain.

1.1 Levels of Natural Language Analysis

Phonology entails interpreting speech sounds within a sentence and individual words, with the goal of converting a speech signal into a textual representation (Rabiner & Schafer, 2007). Phonology is useful to determine the correct interpretation of heteronyms (words that differ in their pronunciation and meaning, but not their writing, e.g., *tear*), or to determine emphasis in a given sentence (Hausser, 2001; Liddy, 2001; Rooth, 1992).

On the **morphology** level, words are broken down into the smallest units of meaning called *morphemes*. Words may be composed of multiple morphemes, and by finding the definitions of the individual morphemes and combining them, one can also discern what certain words mean (Liddy, 2001).

The **lexical** level is about the meaning of individual words in a sentence. At this level, words that only have one meaning may be replaced with semantic representations (Liddy, 2001). Polysemous words cannot be interpreted on the lexical level. These words have multiple definitions and context determines which is the correct interpretation. However, it is

possible to determine the function of the words in a sentence by means of Part-of-Speech (POS) tagging (Liddy, 2001). POS tagging is giving grammatical labels to the words in the sentence. These grammatical labels differ per language. In English, the main categories of words are nouns, verbs, adjectives and adverbs. These classes may contain subclasses, but these are usually not used in POS tagging, as the subclasses may overlap with others, and only 'main classes' are distinguished (Schachter & Shopen, 2007).

On the **syntactic** level, the grammatical structure of a sentence is determined to find the relationship between words. Syntax is important, because the interdependency between words and the order in which they appear conveys meaning (Liddy, 2001; Hausser, 2014). During the syntactical analysis, sentences can be broken down into smaller subsets called phrases. This process is called Phrase Structure Grammar, which is similar to POS tagging. However, whereas POS tagging is about identifying single words, phrases may contain multiple words (Chowdhary, 2020).

Another part of syntax analysis may be to regularize the structure of the sentence. This optional step is meant to simplify the sentence for further analysis. When the structure of the sentence is simplified, further analysis becomes easier. Regularizing the syntax consists of omitting words that are not required for interpretation of the sentence as a whole and turning passive sentences into active ones to make the operator-operand structure clearer (Liddy, 2001).

The goal of **semantic** analysis is to determine what a sentence means and translating this into a structure that is understandable for a computer (Chowdhary, 2020; Liddy, 2001). This includes the disambiguation of words with multiple possible meanings. Using the information on how words interact provided by the syntactical analysis, the most likely definition of a polysemous word is identified (Liddy, 2001). There are multiple ways of achieving this, some of which require information about usage frequency of definitions in certain situations or in general. Other methods consider the local context and yet others use pragmatic knowledge (Liddy, 2001).

Discourse analysis is about the text as a whole, as opposed to just one sentence at a time. It aims at finding the connection between sentences in order to determine the meaning of the text, which is usually broader than the combined meanings of individual sentences (Chowdhary, 2020).

Pragmatic analysis is about the implicit meanings of the text with the goal of determining the intended message, rather than the literal utterances (Lewis, 2013; Liddy, 2001). The information required for pragmatic analysis is not present in the text itself. Rather, almost any information outside the conversation or text may be used and it is up to the receiver to determine which information is relevant, depending on the context (Lewis, 2013).

2. Models on semantic memory

The most challenging aspect of semantic analysis is the need to contain a large amount of data in a structural and accessible manner. In order to determine what certain words or phrases mean in the context of a sentence or text as a whole, the system needs to have a large corpus of information similar to a human's memory.

Our memory contains a large amount of information on any given concept. Even for seemingly basic concepts, people can describe functionality and properties, continuing on with less and less relevant information, such as subtypes, or even other related concepts and its details. Similarly, people have knowledge on a huge number of concepts. For example, the concept of *'pen'* as a writing utensil is different from *'pen'* as a small encasement for animals, and the verb *'to pen'* is yet another distinct concept (Collins & Loftus, 1975). All these concepts and their properties are stored in our memory. There are several models that describe the human knowledge base, most notably feature models and network models.

In a feature model, instances are compared to the target category and based on similarity, the instance is part of the category. In the initial model proposed by Smith, Shoben and Rips (1974), properties of concepts are either *defining traits* or *characteristic traits*. Defining traits are those that are required to describe a concept, whereas characteristic traits are relevant, but not required to define the concept. Processing of new concepts follows a two-stage mechanism; in the first step, all traits of a concept are compared to the target category. When the new concept has enough traits in common with the category it is compared to, it is accepted as being part of said category. When there is reasonable doubt about whether there are enough shared characteristics, the second step of comparison takes place, in which only the defining traits are compared. The new concept is then accepted as part of the category when the defining traits match, even when the characteristic traits are not an exact match (Smith et al., 1974).

One problem with this feature model is the distinction between defining and characteristic traits. For many concepts, it is impossible to distinguish between them, as there are no traits that are inherently required to describe a concept (Collins & Loftus, 1975). For example, one could say that a defining trait of dogs is that they have four paws. However, it is possible for a dog to lose one of its paws, which poses a problem for the feature model. Either a dog that loses one of its paws is no longer an entity belonging to the category *'dog'*, or the trait *'having four paws'* is not required to describe the concept, meaning it cannot be a defining trait. These inferences can be made on many other traits of any concept, leaving no distinctive defining traits to distinguish concepts.

Semantic networks are structures containing nodes representing concepts, and links between them specifying their relation toward each other (Collins & Loftus, 1975; Deliyanni & Kowalski, 1979; Lehmann, 1992). Searching in memory happens through spreading of activation from the input nodes; when a concept is activated, this concept will activate the concepts it is directly linked to. The newly activated concept will do the same for all the concepts it is directly linked to. This is done until an intersection is found between the paths of different inputs. Then, these paths are evaluated to see if they adhere to all the criteria specified by context and syntax (Collins & Loftus, 1975).

Each relation is a weighted connection from one node to the other, meaning concepts may be more or less strongly connected to another concept. The relational links between concepts usually go in both directions, but they may have different connection strength, meaning the association from A to B might be stronger than the other way around (Collins & Loftus, 1975). Therefore, it may take longer to think of B when prompted with A than coming up with A when B is given. The weighted connections will continuously weaken the activation level until it is too low to activate the next node, meaning the memory search does not continue indefinitely (Collins & Loftus, 1975). Aside from the connection weights, the nature of the bidirectional relationship between two concepts can differ as well depending on the starting node; some networks contain different relations like *'actor'* or *'object'* to describe semantic cases, while other networks use relational links such as *'has'* or *'can'* to describe the characteristics of objects (Lehman, 1992).

Quillian (1966, as cited in McClelland & Rogers, 2003) proposed a hierarchical structure for semantic networks, in which concepts may inherit traits from a superordinate concept using *'isa'* links (see figure 1). Therefore, inferences can be made on the concepts on the subordinate level based on the traits of the parent node. For example, a *'robin'* is a bird,

and because of this relation it may be assumed that any traits belonging to *'bird'* also apply to *'robin'*. The appeal of this model is that traits belonging to a whole group of concepts only have to be stored once in memory, at the superordinate level (Chowdhary, 2020; Lehmann, 1992; McClelland & Rogers, 2003).

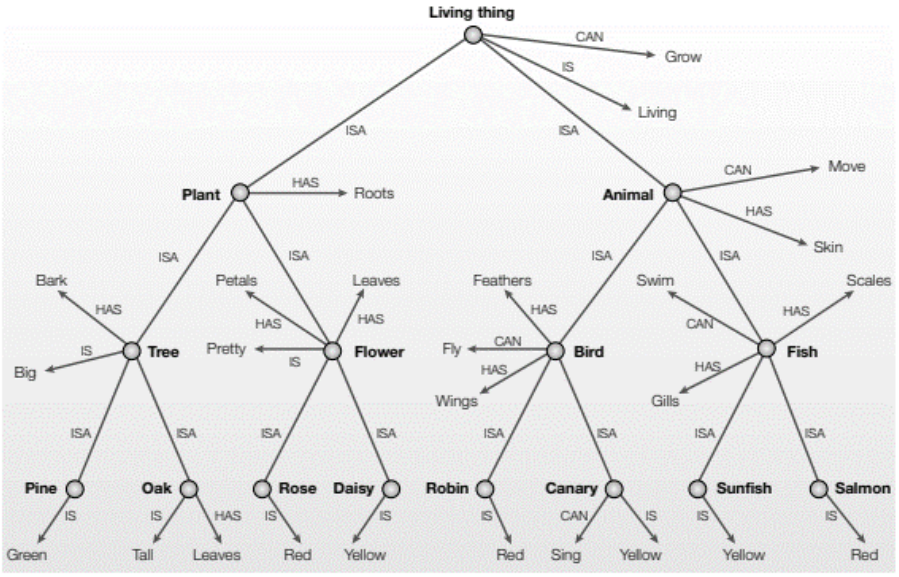


Figure 1: The hierarchical semantic network as proposed by Quillian, adapted to the domain of living things. Arrows represent relational links from a concept to a property. "isa" links determine the hierarchy; any concept that "isa" different concept, inherits all properties of the superordinate category. Figure taken from McClelland & Rogers (2003).

The semantic networks as proposed by Quillian (1966, as cited in McClelland & Rogers, 2003) in which propositions may be generalized for all subcategories also faces some challenges. Firstly, the appeal of the simple hierarchy is simultaneously challenging, as there may be properties that are true for almost all members of a category, but false for others. For example, most plants have leaves, but pine trees have needles instead. The question arises whether to store the property *'has leaves'* at the superordinate level of *'plant'* – which would require storing a negative link to *'leaves'* at all plants that do not share this property – or to store this property at all individual concepts for which the proposition is true, essentially losing the benefit of generalization entirely (McClelland & Rogers, 2003).

Secondly, there are some conflicts between the model and findings of more recent research. If properties were indeed only stored at the superordinate category, it should be expected that people are quicker to name properties unique for any given concept than those that apply to the superordinate category, as the latter are stored further away in memory. For the same reason, it should be expected that information closest to the concept should be

remembered longer than more general information in case of memory loss. However, research shows that these assertions do not hold (McClelland & Rogers, 2003).

McClelland and Rumelhart (1985) adopted some of these ideas in their *Distributed Model of Memory*. This is a semantic network wherein nodes are heavily interconnected and the pattern of activation determines a mental state. Each node has its own role in the sense that any mental state can be activated only by the same sets of nodes every time. Any other combination of active nodes would result in a different mental state and thus a different concept. The network learns by changing the weights of the links between nodes; increasing the weight means more activation is spread from node A to B. Memory retrieval then takes place by cueing part of the information (for example by sensory information), which in turn probes the other nodes required to form the desired pattern (McClelland & Rumelhart, 1985).

Features of this model were implemented by McClelland and Rogers (2003) in their *Parallel Distributed Processing* model. In contrast to the model proposed by Quillian (1966, as cited in McClelland & Rogers, 2003), this model does not impose a strict hierarchy. Rather, it is a multi-layered network, in which all Items and Relations are input nodes connected to a random set of nodes in the network. The nodes in the network are also randomly connected to one or more Attributes in the output layer (see figure 2).

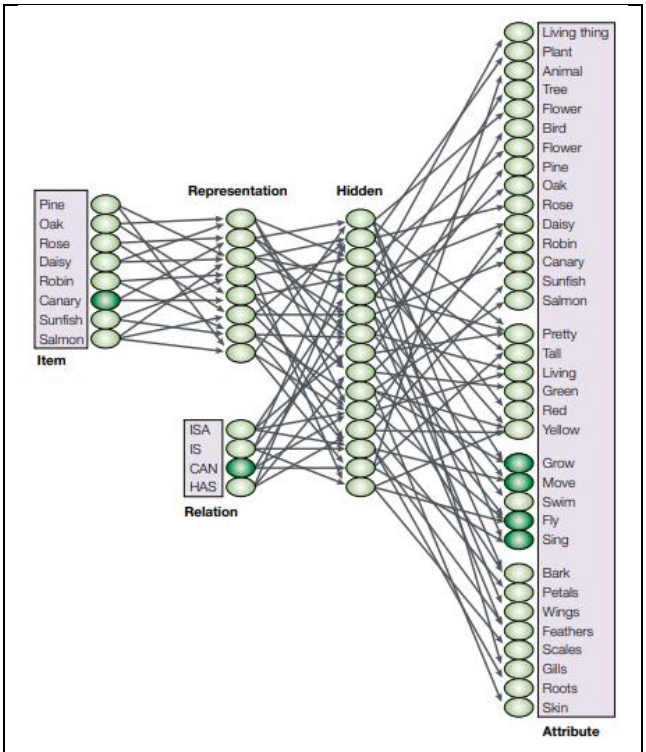


Figure 2: The Parallel Distributed Processing model as proposed by McClelland & Rogers (2003). Figure taken from McClelland & Rogers (2003).

At the start, each item-relation pair is connected to all possible attributes with a low connection weight, and the learning of classification happens by adjusting these weights. Connections that lead to the correct output are strengthened, while connections that lead to a different output are weakened in the process (McClelland & Rogers, 2003). This learning method, called back-propagation (Rumelhart et al., 1986), makes it is possible for the network to draw arbitrarily-shaped classification boundaries.

As certain concepts have overlapping features with other concepts that share a superordinate concept, the gradual reinforcement of correct links in the learning process differentiates general concepts first, before a distinction can be made between specific subcategories. This general-to-specific differentiation process is similar to how humans learn (McClelland & Rogers, 2003; Rogers & McClelland, 2008). The system created by McClelland and Rogers (2003) is capable of finishing three-item propositions. For example, the input activation of *'canary'* and *'can'* results in the activation of *'move'*, *'grow'*, *'fly'* and *'sing'*.

3. Goal of the current research

While the promise of multi-layered networks in terms of semantic networks is great, there is currently no option to run such a type of network with a size even approaching the human knowledge base. It is possible to generate nonlinear classification boundaries using nonlinear projection, but this requires great computational power (Chen et al., 2020). Current computers might be able to run a network that only needs to solve a limited number of problems, but they lack the power for a human-like semantic network. This becomes evident when looking at some of the current applications, such as Apple's *Siri* or Android's *Bixby*; these systems are capable of handling a limited number of commands, but refer the user to a search engine for any prompt outside of their scope.

In current CMOS computers, calculations are made with transistors on a microchip. Increasing the number of transistors on the chip increases the processing power that is available. Moore (as cited in Schaller, 1997) predicted that the number of transistors on a chip would double every 18 ~ 24 months, thereby increasing the computational power of these chips without increasing their size. This prediction was very accurate for the last decades. However, at some point, transistors cannot decrease in size anymore, meaning this way of increasing performance will come to an end (Monroe, 2014; Waldrop, 2016). At some point,

the only way to expand the computational abilities of a chip would be to increase its size, in order to make more room for transistors. As semantic networks inherently require working with a large amount of data, this would mean the chips would have to increase in size drastically, resulting in bigger computer systems and more power consumption. This would be counterproductive when looking at the application domains, such as personal assistants, and it would be a step backward in terms of technological advancement. Therefore, other solutions are required.

One possible solution is to pass on some of the required properties of a semantic network into the hardware, rather than having the software simulate the structure and do all the work. Implementing the structure in the hardware would reduce the number of required calculations, and thereby power consumption and size of the computer. This solution can be found in the domain of neuromorphic computing. Neuromorphic computing mimics the structure of the brain, with the aim of reducing calculation cost and therefore energy consumption. Whereas traditional computers contain one or more processors to deal with information drawn from memory sequentially, neuromorphic computing distributes the memory and calculations among small interconnected units throughout the system. These units represent neurons that are interconnected by synapses (Monroe, 2014). As the structure of neuromorphic computers are closer to that of the human brain, it could be better suited for tasks at which the human brain excels, such as semantics.

Chen et al. (2020) have developed such a hardware solution in the form of a boron dopant cell. This cell contains a ‘reservoir’ of boron atoms doped onto silicon, two input nodes for inputting electrical currents, one output node and five ‘control nodes’ to program the output (see figure 3). The boron atoms in the cell make it possible to conduct one or more input voltages through the cell to the output layer by means of the so-called hopping regime. The control nodes are capable of altering the inner structure of the cell, for example by increasing or decreasing the probability of electricity conduction between atoms. In this manner, the output pattern can be programmed as a non-linear function of the input.

The capabilities of the cell were tested with both linear and non-linear logic gates (Chen et al., 2020). The input pattern is displayed in figure 3b, and the resulting output values can be found in figure 3c. The cell was capable of solving all the logic gates, but the output values were not all equal; for the non-linear gates the output range was tenfold smaller than the output range of the linear gates. This indicates that the non-linear gates are much harder to solve, even when the inner structure can be adjusted using control nodes.

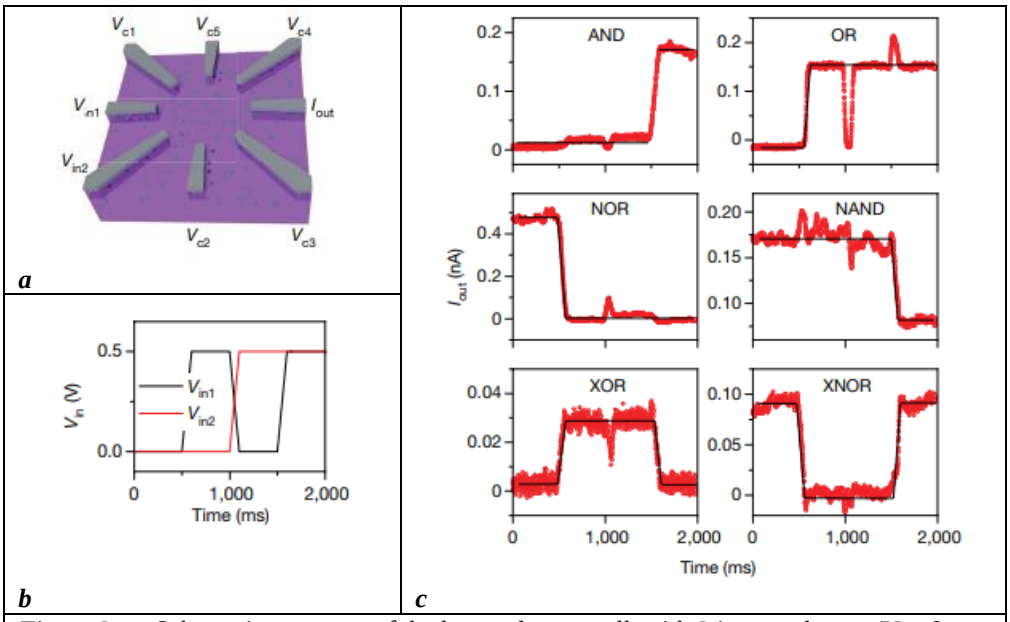


Figure 3: a. Schematic structure of the boron dopant cell, with 2 input voltages (V_{in1} & V_{in2}), 5 control voltages (V_{c1} – V_{c5}) and the output current (I_{out}).
b. The input given to the boron dopant cell over time.
c. Output values for each logic gate given the input depicted in figure b. The output range for linear gates (AND, OR, NOR, NAND) are greater than those of the non-linear gates (XOR & XNOR).
Figures taken from Chen et al. (2020).

Chen et al. (2020) further displayed what the boron dopant cell can do in terms of pattern recognition. Using four input nodes and 16 filters, they were able to do basic number recognition on the Modified National Institute of Standards and Technology (MNIST) digits. The accuracy of the cell was about 96% (Chen et al., 2020).

The goal of the current research is to find out whether a system structured like the boron dopant cell could also be utilized as a basis for a simple semantic network. In doing so, we could establish a baseline for the capabilities of a (physical) reservoir in NLP. Since the actual physical system is not available to work with, we are doing this by simulating the network structure as described by Chen et al. (2020). In particular, the aim is to simulate a boron cell in terms of 'reservoir computing' as described in the next section.

3.1 Reservoir computing

The simulations of the boron-reservoir cell will not incorporate all its physical properties, as the quantum-mechanical behavior is nearly impossible to recreate. Rather, we use a machine learning technique called Reservoir Computing (RC) in place of the hopping regime, which we presume is capable of spreading activation in a similar manner.

RC is a form of machine learning in which a network (also called the reservoir) of recurrently connected nodes is used to transfer an input signal to an output layer. The output

may be programmed by adjusting the connections between the nodes in the reservoir and the output layer (Hinaut & Dominey, 2013). In contrast with feedforward networks, where connections only lead in one direction, connections in recurrent networks may be formed more randomly, allowing for feedback loops (Jaeger & Haas, 2004; see also figure 4). Biological neural networks, such as the human brain, typically also have this property (Jaeger & Haas, 2004). Recurrent connections provide some form of non-linearity, as the relatively simple input pattern is projected onto a high-dimensional network, increasing the separability of the input (Hinaut & Dominey, 2013; Verstraeten et al., 2007).

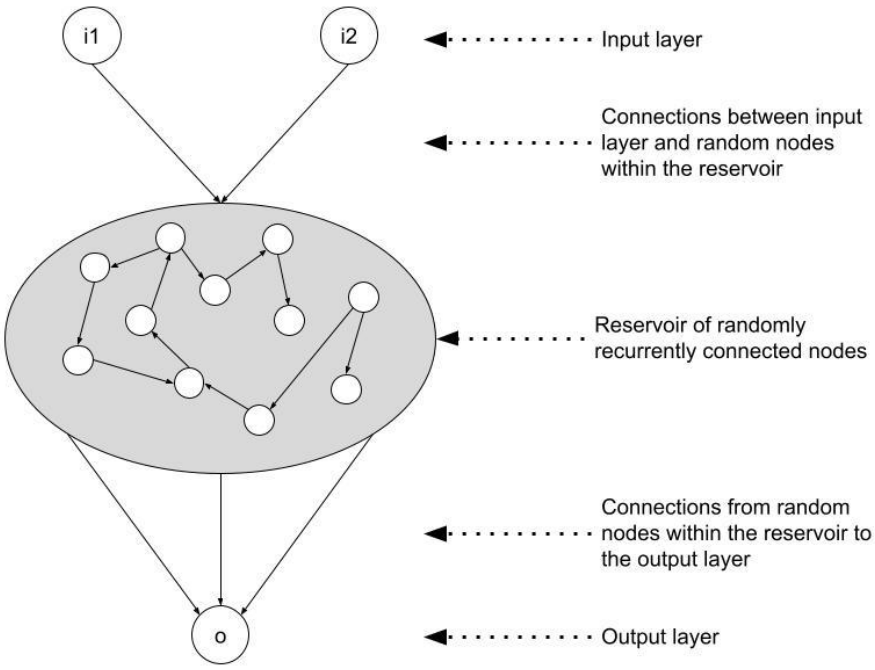


Figure 4. Schematic representation of a recurrent neural network. The input layer is connected to the reservoir. Within the reservoir, there are random, recurrent connections between nodes. This allows for pathways to form, which are in turn connected to the output layer.

RC methods initially did not get a lot of traction, mostly due to ineffective learning methods (Verstraeten et al., 2007). Typically, these methods entail adjusting connection weights between all nodes within the reservoir and the connections from the reservoir to the output layer. Due to the number of nodes and therefore possible connections within a reservoir, this process would result in slow convergence (Jaeger & Haas, 2004; Verstraeten et al., 2007).

However, more recent research has found other learning algorithms that do not require adjusting internal connections. One such algorithm was developed by Jaeger and Haas (2004), namely Echo State Networks (ESN). The ESN approach makes use of a relatively large

reservoir (up to 1000 neurons) and sparse interconnectivity (1%). Rather than adjusting all connection weights in the reservoir, the algorithm only trains the weights from the reservoir to the output node. In this manner, it is a simple linear regression model, decreasing the time taken per learning run to only a few seconds to minutes depending on reservoir size (Jaeger & Haas, 2004). A similar approach was coined by Maass et al. (2002), the Liquid State Network (LSN). The LSN also contains a recurrent circuit that learns by adjusting the readout layer rather than the entire network.

The structure of ESNs or LSNs are similar to biological neural networks. They contain a large number of neurons, that are sparsely and randomly connected to form recurrent pathways from an input layer through the reservoir to an output layer (Jaeger & Haas, 2004). This makes the structure fitting to simulate the boron dopant cell as described by Chen et al (2020), as the boron cell essentially functions in a similar way. Input activation is given to a 'reservoir' of boron atoms, and spread through the cell by means of the hopping regime until it reaches the output layer.

The current research is focused on the difference in learning algorithms between ESNs / LSNs and the boron dopant cell designed by Chen et al (2020). Whereas ESNs and LSNs still require adjustment of connection weights, the boron dopant cell makes use of control nodes to program the output.

3.2 Research questions

The goals of the current research are to simulate the structure of the boron dopant cell as developed by Chen et al (2020) as a reservoir and to determine its capabilities of running a small-scale semantic network.

In order to do this, we first test its ability of solving logic gates using perceptron learning. This way, we can test the impact of adding a reservoir to basic perceptron learning. The second step is replacing the perceptron learning algorithm by implementing five control nodes to program the output. Finally, the reservoir with control nodes will be adjusted to be used for implementing the Parallel Distributed Processing model of semantic relations (McClelland & Rogers, 2003).

Although we are not mimicking the quantum-mechanical aspects of the physical boron dopant cell, the overall functionality and structure does match. The reservoir is capable of transferring an input through a hidden layer to the output node, much like how the physical cell is capable of conducting a current through the boron atoms to the output node. Therefore,

some meaningful inferences about how the physical cell could work can be made based on these simulations. Furthermore, the implementation of control nodes could provide a viable alternative to using perceptron learning.

Using simulations rather than a physical cell is a low-cost alternative to testing with an actual boron dopant cell, as these are hard to create and not readily available. Simulations also allow for more variations in aspects of the cell (as explained in section 4.1).

4. Model description

The structure of our boron-reservoir cell was modeled after the boron dopant cell as described by Chen et al. (2020), i.e., a network consisting of an input layer, a middle cell layer (the reservoir cell) and an output layer. In this network, each input node is randomly connected to a number of nodes in the reservoir. The nodes in the reservoir are also randomly interconnected, and a random subset of nodes in the reservoir is in turn connected to the output layer. As outlined in section 3.1, this allows for certain 'pathways' to form that make it possible to spread activation from the input nodes through the middle layer to the output node.

Activation onto each node in the reservoir is calculated by summing the weighted input from the respective input nodes. The summed value is then the input to an activation function, which typically squashes the value into a small output range (Van der Velde, 2020). Two commonly used squashing functions are the logistic function – which results in an activation between 0 and 1 – and the hyperbolic tangent function – which returns a value between -1 and 1. An input of 0 will result in the middle ground for both of these functions, i.e., ½ for the logistic function and 0 for the hyperbolic tangent. In our simulations, we make use of the hyperbolic tangent function, to ensure that an input activation of 0 will also result in an output activation of 0.

Output, in turn, is calculated by summing the weighted activation from each cell node in the reservoir that is connected to the output node. This value is not squashed to prevent information loss. Spreading of activation through the reservoir happens sequentially in a predetermined amount of timesteps.

The network is trained to solve the linear logic gates AND, NAND, OR and NOR, as well as the non-linear logic gates XOR and XNOR. As input, we use different combinations of 1 and 0. The input patterns, as well as the expected outcome for each logic gate, can be found in table 1. These output values are simplified, as the network is not trained to tune to

these exact numbers. Rather, the output value 1 means the activation should be larger than a certain threshold, whereas output value 0 means at or below the threshold. How this threshold is determined is explained in section 4.2.

Table 1: Input patterns and expected output for each logic gate. The input [X, Y] means that the first value is given to the first input node, and the second value is given to the second input node.

Input [X, Y]	***AND***	***NAND***	***OR***	***NOR***	***XOR***	***XNOR***
[1, 1]	*1*	*0*	*1*	*0*	*0*	*1*
[1, 0]	*0*	*1*	*1*	*0*	*1*	*0*
[0, 1]	*0*	*1*	*1*	*0*	*1*	*0*
[0, 0]	*0*	*1*	*0*	*1*	*0*	*1*

The simulations were made in Python v3.7, using the libraries 'numpy', 'math', 'random' and 'xlwt'. The complete source code of every iteration can be found in appendix B. The network is represented by multidimensional numpy arrays; one for storing connections from each input node to the cell nodes in the reservoir, one for storing the connections from each cell node to all other cell nodes in the reservoir, and another one for the connections from each cell node to the output node. Another multidimensional array is used to store the activation levels of each cell node in the reservoir at every timestep for every given input. The math library is used for the squashing function (tanh). The library 'random' is required for generating the connection matrices, as connections between nodes are to be determined randomly. Lastly, xlwt is used for generating the output files as Microsoft Excel files.

4.1 Parameters used to generate the reservoir

Although the connections between nodes are generated randomly (i.e., which connections are formed between the input nodes and the reservoir, within the reservoir and between the reservoir and the output node), certain aspects of the cell are controlled for in these simulations; cells size, sparsity, connection strength and seeds. These aspects each influence the capabilities of the reservoir to transfer the input through the cell layer to the output node. Aside from these parameters, two additional parameters are used for the learning functions, namely the maximum number of learning runs and the learning rate used for adjustments. For these parameters, we used constant values based on results of initial testing (see appendix A – supplementary methods & results), namely 1000 learning runs and a learning rate of 0.01.

Cell size stands for the amount of cell nodes within the reservoir. Increasing the cell size also increases the potential number of connections between nodes. More connections lead to more possible pathways from input, through the reservoir to the output node. When the

reservoir contains too little nodes, it is less likely that activation given into the cell reaches the output node.

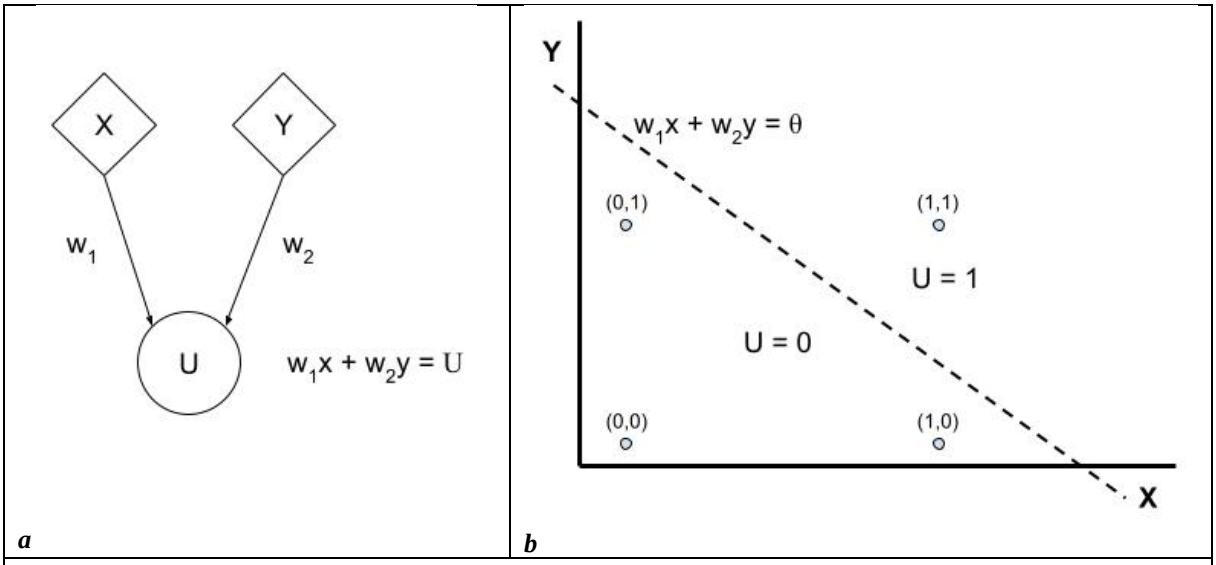
Sparsity determines how many connections are formed between nodes at all layers (i.e., from input to cell, between cell nodes and from cell to output). Similar to cell size, more connections lead to a higher possibility of input reaching the output node. Sparsity is used as a threshold for deciding whether a connection is formed between nodes. For each potential connection, a random number between 0 and 1 is generated (see below), and a connection is formed only when this number exceeds the sparsity (threshold). Therefore, higher sparsity means fewer connections are formed. The generated number is not used as connection strength between nodes, but is only used for determining whether a connection is formed or not.

Each connection between nodes has an initial weighted value, which is set by the **connection strength**. The activation of each node is multiplied by this connection strength when it spreads activation to the next node, before the summing or squashing takes place at the receiving node. Connection strength between nodes influences the overall activation of the cell, as a higher connection strength generally results in higher activation in the post-connection node.

Whereas cell size and sparsity influence the number of possible connections, the **seed** impacts which connections are formed. Drawing completely random numbers is not possible for computer systems. Rather, when asking for a 'random' number, the system reads numbers from a large database. The seed determines the starting point for drawing these pseudo-random numbers. Using a different seed results in different numbers being drawn for testing against the sparsity threshold, meaning different connections will be formed.

4.2 Perceptron learning

As a baseline for the capabilities of the boron cell as a reservoir, we will initially use a simple machine learning algorithm called perceptron learning (Rosenblatt, 1958). Perceptron learning is based on the Hebbian learning rule of positive or negative reinforcement influencing pathways of neurons in our brain (Hebb, 1930, as cited in Brown, 2020). In its most basic form, a perceptron consists of two input neurons and one output neuron. Each input neuron is connected to the output neuron with a weighted connection, which is a scaling factor for the input activation (see figure 5a).

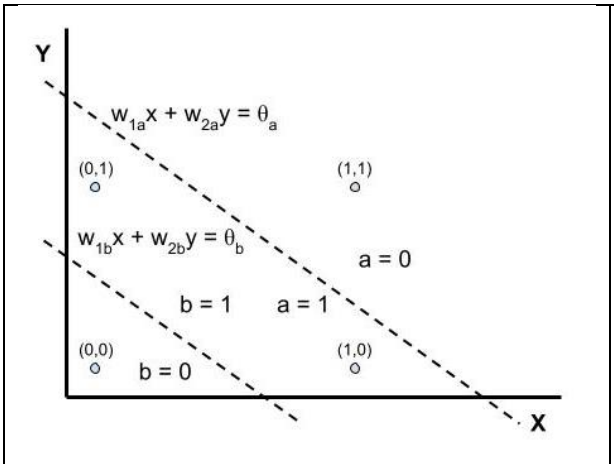


a ***b***

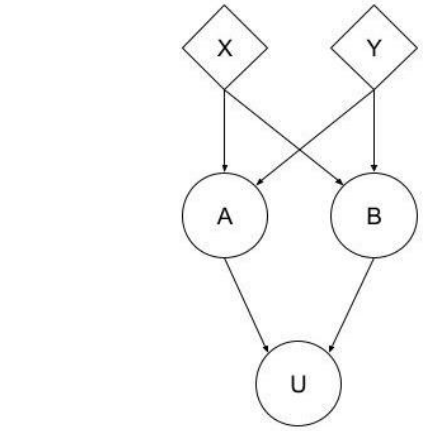
Figure 5: a. Depiction of a basic perceptron, consisting of two inputs (X and Y), one output node (U) and weighted connections (w_1 and w_2 for X and Y, respectively). Output is calculated by summing the weighted activations from input X and Y.
b. The threshold line determining the output activation for node U. When the function $w_1x + w_2y$ exceeds the threshold, activation of output node U will be 1, otherwise it will be 0. The perceptron learning algorithm adjusts this threshold (bias) to ensure the output matches the expected pattern.

The receiving output node is only activated when the total amount of input exceeds a certain 'activation threshold'. When this threshold is positive, a larger positive activation onto the node is required for it to fire. Conversely, when the threshold is negative, the node needs to receive a larger negative input activation to output 0. By adjusting both the connection weights and the activation threshold, it is possible to program the output behavior of any node (see figure 5b). This way, it is possible for the perceptron to solve any linearly separable logic gate (Van der Velde, 2020).

Basic perceptron learning is not suitable to classify the XOR or XNOR gates, because it is impossible to draw a classification line that divides the points [1,0] and [0,1] from [0,0] and [1,1] (see figure 5b). In order to solve these logic gates, there is a need for non-linear projection. This can be done by adding another layer between the input nodes and the output node of the perceptron that performs intermediary classifications. When adding these outputs together, the network should be capable of solving non-linear problems as well (see figures 6a and 6b). For the XOR gate, the intermediate layer should solve for AND and NOR; for the XNOR gate, the gates to be solved are NAND and OR (see figure 6c).



a



b

XOR

X	*Y*	*A (OR)*	*B (NAND)*	*A + B*	*U (XOR)*
1	*1*	*1*	*0*	*1*	*0*
1	*0*	*1*	*1*	*2*	*1*
0	*1*	*1*	*1*	*2*	*1*
0	*0*	*0*	*1*	*1*	*0*

XNOR

X	*Y*	*A (NOR)*	*B (AND)*	*A + B*	*U (XNOR)*
1	*1*	*0*	*1*	*1*	*1*
1	*0*	*0*	*0*	*0*	*0*
0	*1*	*0*	*0*	*0*	*0*
0	*0*	*1*	*0*	*1*	*1*

c

Figure 6: a. Drawing one classification line is not sufficient to divide points [0,0] and [1,1] from [1,0] and [0,1]. Therefore, non-linear projection is required. This enables drawing multiple classification lines; one to isolate [1,0] and one to isolate [0,0].
b. Depiction of a multi-layered perceptron. Input X and Y are given to both subcells (A and B). Node A solves a linear gate, and node B solves another by adjusting the connection weights from [X,Y] onto A or B, respectively. When taking their outputs together at the output node (U), it should be able to solve the non-linear logic gates XOR and XNOR by adjusting the bias.
c. Overview of intermediate gates to be solved for XOR and XNOR. For XOR, taking the sum of A and B is not yet sufficient to reach the XOR gate. Activation levels have to be lowered to reach the correct pattern. For XNOR, this lowering of activation is not required.

5. Experiment 1:

Perceptron learning with a single reservoir

The first program contains a single reservoir of *n* nodes. The input [X, Y] (see section 4, table 1) is given to a random subset of nodes within the reservoir. The activation is spread through the reservoir and reaches output node U, before the basic perceptron learning algorithm takes place. The learning function adjusts all the output connections and recalculates the output by summing the activation level of cellnodes multiplied by the adjusted output connections and adding the bias value (see figure 7). Changing the output connections does not influence the

activation levels within the reservoir, but the output activation is altered when these weights are adjusted.

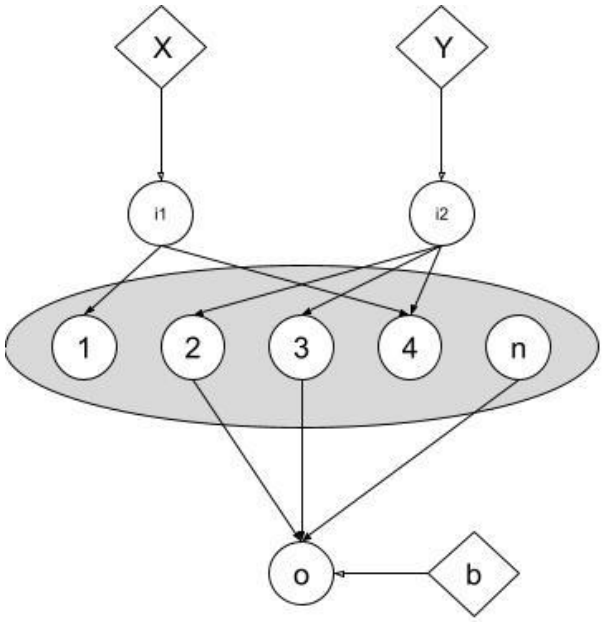


Figure 7: Structure of the reservoir as implemented. Input [X, Y] is given to a random subset of nodes in the reservoir (depicted as grey oval) by i1 and i2, respectively. Nodes in the reservoir are interlinked randomly (connections not drawn for clarity of the picture), and a random subset is connected to output node O. Output activation is calculated by summing the weighted activation of all cell nodes connected to the output node and adding the bias value. The perceptron algorithm adjusts the connections from the cell nodes to the output node, as well as the bias (b).

Although there is no explicit non-linear projection in the current program, it is possible that the existence of *n* nodes in the reservoir is already sufficient to induce some non-linearity, as was the case for the dopant cell by Chen et al. (2020). Therefore, we are testing all logic gates to determine if the reservoir is capable of spreading activation and whether the activation levels can be programmed to solve for all logic gates. Table 2 provides an overview of the parameter values used to generate the reservoir. Each combination of parameters has been tested, resulting in a total of 480 simulations.

Table 2: Overview of parameters used for creating the cell in experiment 1 (perceptron learning with a single reservoir). Each combination of values was tested.

Cell size	*10*	*20*	*50*	*100*	-	-
Sparsity	*0.80*	*0.85*	*0.88*	*0.92*	*0.95*	*0.98*
Connection strength	*0.2*	*0.5*	*0.8*	*Random*	-	-
Seeds	*1, 2 ,3*	*4, 5, 6*	*7, 8, 9*	*10, 11, 12*	*13, 14, 15*	-

Results

The overall success rate of the current program can be found in figure 8. Here, 'Success' means all linear and non-linear logic gates were solved with the same parameter settings,

'Partial' means one or more (but not all) logic gates were solved, 'Failure' means none of the logic gates reached convergence using a certain configuration of parameters. The success rate per logic gate can be found in table 3.

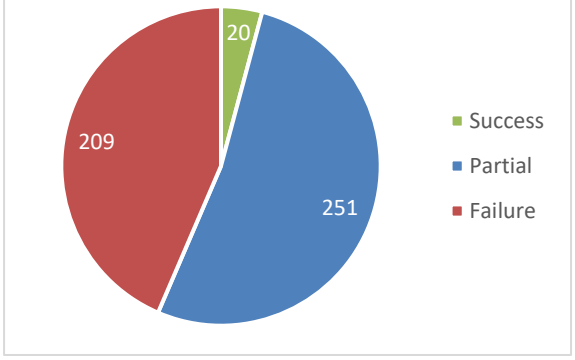


Figure 8: Success ratio of all logic gates using perceptron learning with a single reservoir.

Table 3: Success rate per logic gate for perceptron learning with a single reservoir

Logic gate	***Success rate***
OR	*271 (56.5%)*
NOR	*247 (51.5%)*
AND	*113 (23.5%)*
NAND	*88 (18.3%)*
XOR	*23 (4.8%)*
XNOR	*37 (7.7%)*

Generally, the reservoir appears to add complexity to the perceptron algorithm, resulting in a low success ratio even for the linear logic gates. The basic perceptron always reaches convergence for linear gates, but using the reservoir, the highest success rate is reduced to 56,5%. This indicates that more than 40% of the configurations do not allow for activation flow from input- to output nodes. The complexity of the cell might result in too few connections being formed, or at the very least that there is a lack of complete pathways from the input nodes through the reservoir to the output node. Appendix A gives a more detailed explanation on how each parameter used for generating the reservoir impacted the number of pathways and, by extension, the success ratio.

There is a notable difference in the capabilities of solving the OR / NOR logic gates compared to AND / NAND. The OR / NOR gates get solved by more than half of the cell configurations, whereas AND / NAND are only solvable by a quarter of them. This difference can be explained by the difficulty of determining the threshold. For OR / NOR, the classification line only needs to differentiate between "activation" versus "no activation". Any activation level above 0 that reaches the output node should result in a positive activation for OR, and a negative (or no activation) for NOR and vice versa. This is already achievable when there is at least one complete pathway from input to the output node.

For the AND / NAND gates, the perceptron algorithm needs to find a threshold that differentiates the output for [1,1] from the others. This requires more nuance, as multiple input patterns may result in a positive activation. It is generally expected that an input of [1,1]

results in a higher activation level than [1,0] and [0,1], but the complexity of the network may result in activation levels that are approximately equal. This could happen when there are too many connections, because then most, if not all, nodes within the reservoir get a maximum activation for any input. Alternatively, when there are not enough connections, it could happen that complete pathways are formed only between one input node and the output node. In these cases, activation for [1,1] will not exceed other activation levels. Therefore, lowering all activation levels does not solve the AND / NAND gates.

Another notable difference can be found between OR / AND versus their NOT counterparts (NOR / NAND, respectively). For these logic gates, the perceptron algorithm first has to adjust the connection weights to a negative number first, and then adjust the bias in such a way that the threshold is set properly. As the bias is adjusted concurrently with the connection weights, it is possible that it needs more adjustments after the connection weights have been set to a negative number. Furthermore, it is possible that the number of connections to the output node influence this process, because the learning algorithm adjusts all connection weights simultaneously. Each adjustment could impact the resulting activation greatly, meaning the bias has less impact overall.

Interestingly, the reservoir is capable of solving the non-linear logic gates with some configurations. It appears the random connections within the reservoir could lead to a multi-layered structure, even when a 'hidden layer' is not explicitly programmed. Although the success rate is low, it does show that the reservoir complicates the basic perceptron algorithm. In future iterations, we will explore a structure in which the hidden layer is more explicitly programmed.

6. Experiment 2:
Perceptron learning with multiple reservoirs

To allow solving intermediate logic gates, the use of 'subcells' with their own input-to-output streams was introduced, as an additional layer (see figure 9). These reservoir subcells are comparable to more densely connected clusters of nodes in a physical reservoir. Both subcells receive the same input [X, Y] as in experiment 1 and would behave similar to the single reservoir in the first program. For the linear logic gates, this means that both subcells could try to solve the same linear gate, whereas for the non-linear gates, the subcells could perhaps try to solve the intermediate gates as described in figure 5c.

In this version, we are still making use of the perceptron learning algorithm for both subcells. However, at each learning run, the summed output is calculated and an overall bias is added to check whether these values solve the 'overarching' logic gate. Therefore, it is possible that the summed outputs of both subcells solve the logic gate, even when one or both cells have not yet reached convergence.

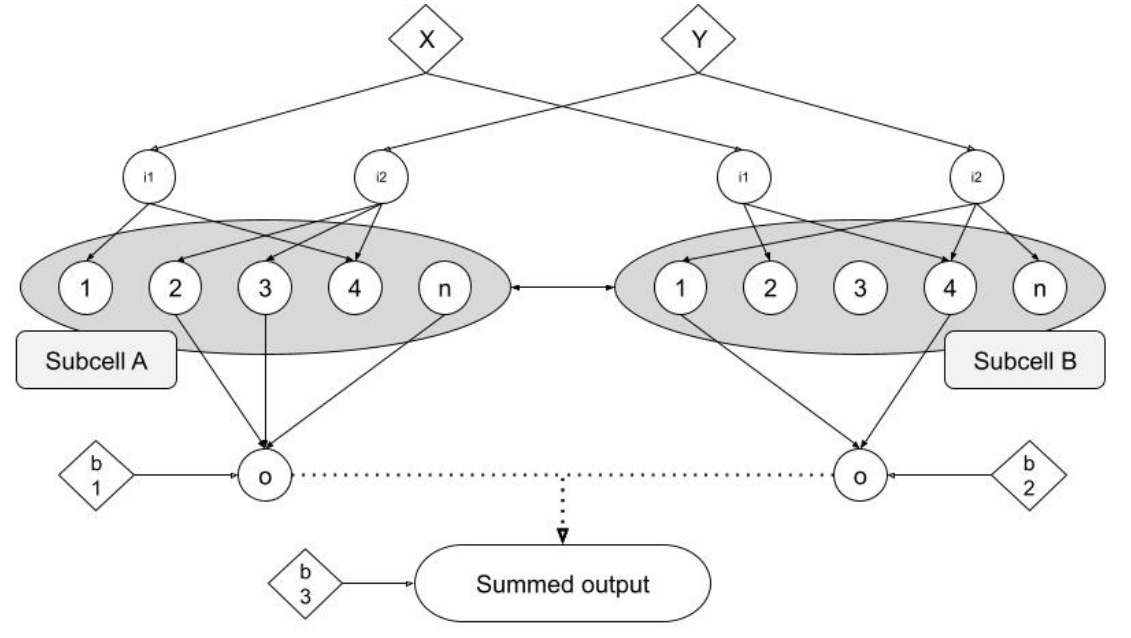


Figure 9: Structure of the reservoir as implemented. Input [X, Y] is given to a random subset of nodes in both reservoirs (Subcell A & Subcell B). Nodes within the reservoirs are interlinked randomly, and a random subset of nodes in one subcell is connected to the other. In each reservoir, a random subset of nodes is connected to output node O. Output activation of each subcell is calculated by summing the weighted activation of all cell nodes connected to the output node and adding the bias value. The perceptron algorithm adjusts the connections from the cell nodes to the output node, as well as the bias (b1 and b2 for subcell A and B, respectively). The activations of both subcells are combined with another bias value (b3) to check whether the 'overarching' logic gate is solves.

In a physical reservoir, it is possible that clusters of nodes still have some connections to other clusters of nodes. To simulate this, we have added connection matrices from one subcell to the other and vice versa. This way, we can determine whether the perceptron algorithm would still be able to succeed even when the activation of the subcells influence each other. The number of connections made between subcells is governed by a new parameter; **external sparsity**. This parameter functions in the same way as sparsity, but we used different values to test a broader range (0.20 for a large number of connections to 1.00 for no connections between subcells).

Other parameters that were present in the previous version were used in a slightly different manner; cell size now determines the size of each subcell, making the reservoir as a

whole twice as big. The seeds were altered to also include a seed for connections between cells. Table 4 provides an overview of the values used for these parameters in this version. Both subcells were generated using the same values, with the exception of seeds. For this parameter, all combinations were tested (e.g., [1,2,3,4] for subcell A and [5,6,7,8] for subcell B and vice versa). Using all combinations of parameters, the program ran a total of 9216 simulations.

Table 4: Overview of parameters used for creating the subcells in experiment 2 (perceptron learning with multiple reservoirs). Each combination of values was tested.

Cell size	*10*	*20*	*50*	*100*	-	-
Int. sparsity	*0.80*	*0.85*	*0.88*	*0.92*	*0.95*	*0.98*
Ext. sparsity	*0.20*	*0.50*	*0.85*	*0.90*	*0.95*	*1.0*
Connection strength	*0.2*	*0.5*	*0.8*	*Random*	-	-
Seeds	*1, 2, 3, 4*	*5, 6, 7, 8*	*9, 10, 11, 12*	*13, 14, 15, 16*	-	-

Results

Figure 10 shows the overall success rate of the current program. As with the first program, "Success" indicates all logic gates converged with the same configuration, "Partial" means one or more converged, and "Failure" means no convergence was reached. A comparison of the success rate per logic gate between the current program using subcells and the single reservoir program can be found in table 5 (results of single cell are taken from experiment 1, table 3). More details on the configurations that made the linear logic gates and the XOR / XNOR work can be found in appendix A.

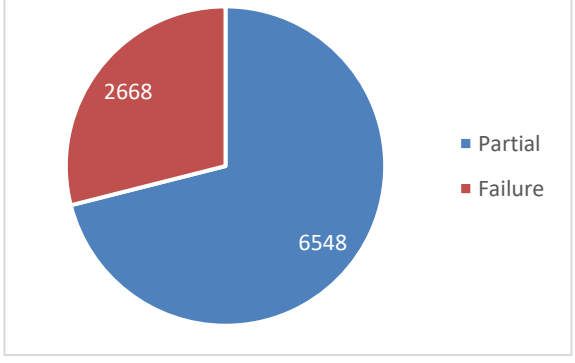


Figure 10: Success ratio of all logic gates using perceptron learning with multiple reservoirs.

Table 5: Success rate per logic gate for subcells (current structure) vs single cell (taken from experiment 1).

Logic Gate	***Subcells***		***Single cell***	
OR	*6548*	*(71,1%)*	*271*	*(56,5%)*
NOR	*6446*	*(69,9%)*	*247*	*(51,5%)*
AND	*1364*	*(14,8%)*	*113*	*(23,5%)*
NAND	*1060*	*(11,5%)*	*88*	*(18,3%)*
XOR	*11*	*(0,1%)*	*23*	*(4,8%)*
XNOR	*13*	*(0,1%)*	*37*	*(7,7%)*

Using multiple reservoirs, there are no configurations of the cell that solve all logic gates. The main reason for this is the low number of times the XOR / XNOR solved, leaving only a maximum of 11 configurations that solve for all logic gates. The 11 configurations that

made XOR work were different from the 13 configurations that worked for XNOR, making complete successes impossible.

The success rate for AND / NAND dropped significantly as well. This could be the result of combining configurations that did not work as a single reservoir with other configurations. When the subcells interact with each other, it is possible that the unfit subcell impacts the activation in the second subcell, interfering with its capability to solve.

Compared to the single cell version, the OR / NOR logic gates have a higher success rate, which also leads to a higher partial success rate overall. This is likely due to the difference in difficulty of solving each logic gate and the combinations of subcell configurations. For OR / NOR, more than half of the previous cell configurations already reached convergence. This means that it is more likely to combine two 'successful' configurations than it is to combine two 'unsuccessful' ones, which leads to a higher success rate. Additionally, receiving input from the other subcell actually benefits the success rate, because it could mean the activation from one subcell reaches the output node of the other subcell even when the input given to the latter subcell would otherwise be lost due to dead ends in the cell.

It seems that solving the intermediate gates is not enough to solve for the non-linear gates using perceptron learning. Rather, using multiple reservoirs with this learning algorithm only further complicates the solving of logic gates. Still, the structure containing multiple reservoirs is better suited for implementing control nodes than the single reservoir, as it makes it possible to more precisely dictate which nodes in the cell are affected by the control node activations. Moreover, one might assume that different patches of boron dopant atoms would exist in the physical boron cell as well, rather than a homogenous distribution of dopants. In the next version of the program, the control nodes are implemented in a structure similar to the current version.

7. Experiment 3: Learning with control nodes

The next step in simulating the boron system by Chen et al. (2020) as a reservoir system consisted of introducing control nodes for learning. Instead of only influencing the output connections, the control nodes in this system have three different functions: 1. reversing the input from positive to negative (C1 / C2); 2. adding or reducing activation onto each cellnode

(C3 / C4); and 3. adding or reducing activation onto the summed outputs of the subcells (C5, see figure 11). In other words, they either influence the 'sign' of activation (as in the sign of the current in an actual boron cell) or the level of activation (current).

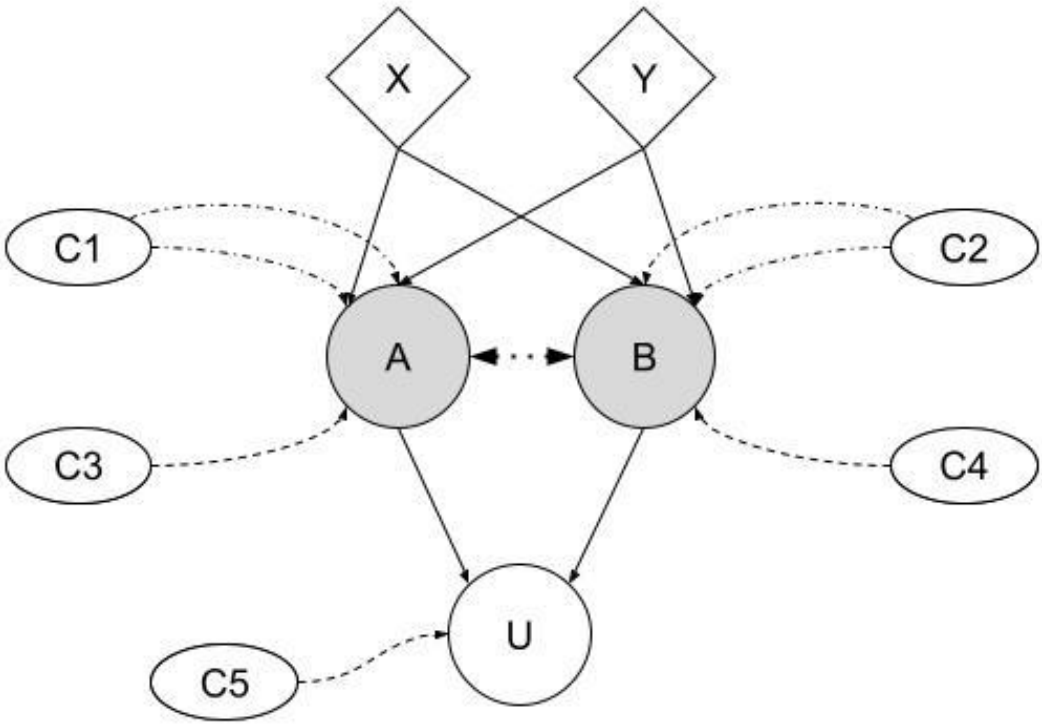


Figure 11: Depiction of the simulation of the dopant cell with control nodes. Input X and Y are given to both subcells (A and B). Like the previous iterations, these subcells contain n *interconnected nodes and are connected to each other. C1 and C2 influence the input connections of subcells A and B, respectively. C3 and C4 add activation into the subcells. C5 adds activation at the output level.*

The first type of control nodes (C1 / C2) is there for solving the NOR and NAND gate more efficiently. By reversing the input to a negative number instead of a positive one, input [1,1] now gives the lowest activation in the cell instead of the highest. Therefore, the output connections no longer have to be adjusted multiple times to get a negative activation when any input is given into the cell, thus sharply reducing the number of learning runs required for reaching convergence.

The second type of control nodes (C3 / C4) adds or reduces activation onto each node in the cell, before the activation function. Therefore, these nodes do not adjust any connection weights, but rather influence the activation level in the cell itself. Due to the complexity of the network, the additional activation is spread non-linearly throughout the network before reaching the output node. The last control node (C5) operates in a similar way to the bias in basic perceptron learning; it adds or subtracts from the summed outputs of both subcells until the correct threshold is reached.

Most parameters used for generating the reservoirs function similarly in this iteration as in the subcell version of the program. However, due to time constraints and the low variation of success rate for different seeds in the previous version, we decided not to adjust for this parameter in the current and upcoming iterations. Rather, we gave both subcells just one combination of seeds for every configuration; subcell A was generated with seeds [1, 2, 3, 4] and subcell B with [5, 6, 7, 8]. In total, 576 variations were tested.

Results

The overall results can be found in figure 12, the success ratio per logic gate for the current structure (control nodes) contrasted to the structure from experiment 2 (perceptron learning; see also table 5) is presented in table 6. Like the 'Perceptron with multiple reservoirs' version, the current version shows no configurations capable of solving all logic gates. The partial success and failure rates are approximately equal. A more detailed report of the effect of each parameter on the success ratio of the linear gates as well as the XOR / XNOR gates separately can be found in Appendix A.

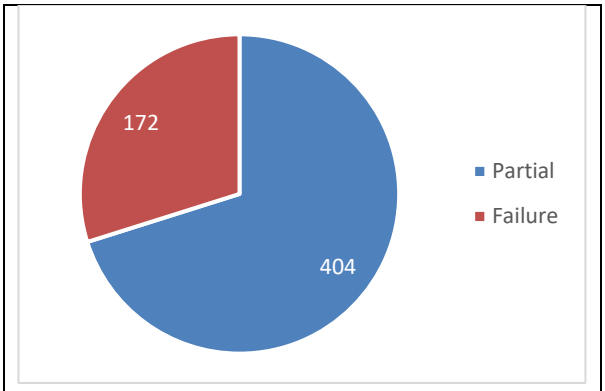


Figure 12: Success ratio of all logic gates using control nodes for learning.

Table 6: Success rate per logic gate for learning with control nodes vs perceptron learning with subcells.

Logic Gate	***Control nodes***		***Perceptron***	
OR	*404*	*(70,1%)*	*6548*	*(71,1%)*
NOR	*403*	*(70,0%)*	*6446*	*(69,9%)*
AND	*242*	*(42,0%)*	*1364*	*(14,8%)*
NAND	*245*	*(42,5%)*	*1060*	*(11,5%)*
XOR	*23*	*(4,0%)*	*11*	*(0,1%)*
XNOR	*16*	*(2,8%)*	*13*	*(0,1%)*

OR / NOR show a similar success rate to the perceptron learning version. This is expected, because OR should be solvable without adjustments of any kind and NOR only requires the connection weights to be negative. Using the first set of control nodes (C1/C2), this process is a lot quicker. The NOR gate always converges on the second learning run due to the reversing of the input connections, while the perceptron algorithm requires multiple adjustments to the connection weights in order to shift them to a negative number. Control nodes do not increase the success rate of these logic gates, as the number of configurations that do not have pathways from the input nodes to the output node do not differ. Adding activation into the cell using control voltages will therefore not result in different activation patterns for input [0,0] compared to any other input.

For the AND / NAND gates, the control nodes show a drastic performance increase. This suggests that injecting activation into the cell has a more nuanced effect on the total activation than adjusting the output connections, making it possible to get different activation levels for input [1,1] as compared to other input patterns with more configurations of the cell. Aside from the increased performance, it seems that control nodes are better suited for solving both logic gates. Whereas perceptron learning solved for AND more often than NAND, the success rates for these logic gates are approximately equal when using control nodes.

The XOR / XNOR gates also appear to be more successful using the control nodes as opposed to perceptron learning with multiple reservoirs. However, the success rate is still relatively low compared to the linear logic gates. Furthermore, there were no configurations of the cell in which both XOR and XNOR solved. It seems that the more nuanced effect of control nodes is beneficial, but not sufficient, to make the non-linear gates work consistently.

8. Experiment 4:
Splitting input streams

In order to further increase the success rate, it could be beneficial to split the input streams given to the subcells, and to remove the connection between subcells. Given that the cell was capable of solving mostly when there were little to no connections between subcells (see appendix A), it is reasonable to remove these connections entirely. Furthermore, splitting the input streams should improve the nuance provided by the control nodes, as the input X will not influence input Y as much, meaning the activation levels will look different.

Splitting the input streams is done by splitting up subcells A and B into a total of four reservoirs; A_1 and A_2 aimed for solving one logic gate, B_1 and B_2 aimed to solve another. Then, the activation for A (A_1 and A_2) and B (B_1 and B_2) are summed up together again to solve the overall logic gate. In order to test whether splitting input streams benefits the success rate, we tested the same external sparsity settings we previously used for connections between cells (in experiments 2 and 3) for connections between input streams. The structure of this version is as seen in figure 13.

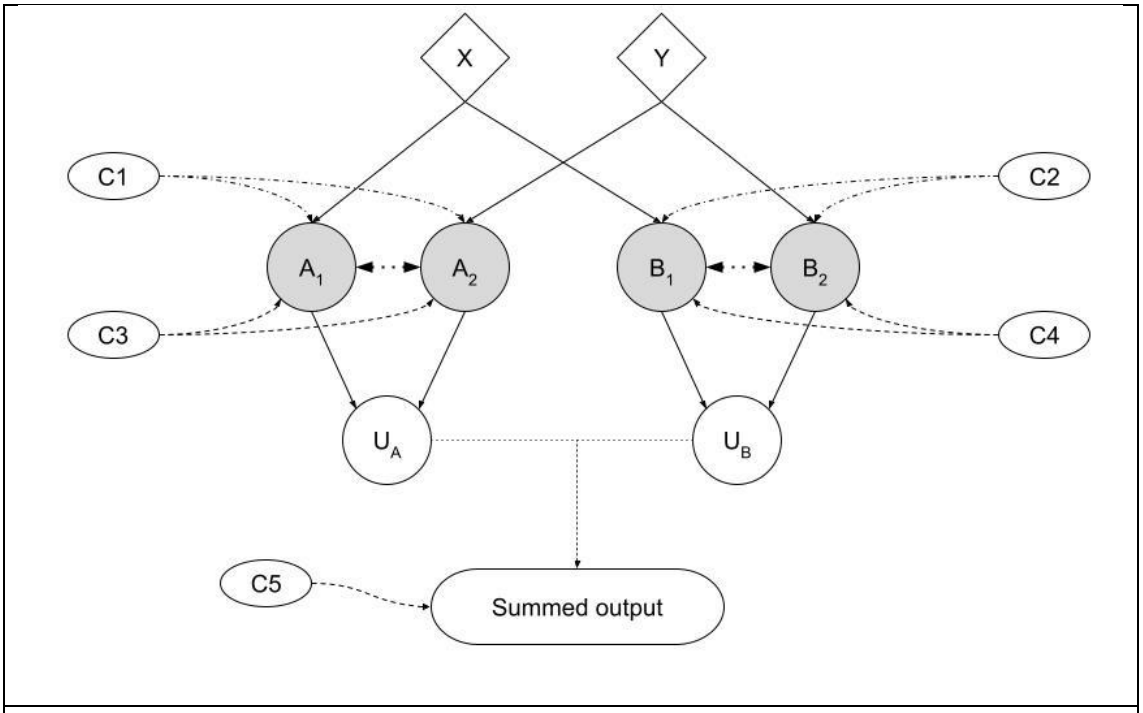


Figure 13: Depiction of the simulation of the dopant cell-reservoir with control nodes and separate input streams. Input X is given to A_1 and B_1, and Y to A_2 and B_2. All control nodes still behave in the same manner.

The control nodes still behave in the same manner as before, but the influence of C1 – C4 is now limited to one intermediate gate. C1 and C2 only affect the activation level in A, whereas C3 and C4 only impact B. C5 still adds onto the summed activation of the cell. The values for the parameters also remained unaltered. However, cell size now determines the size of a subcell (e.g., A_1 representing one input stream), internal sparsity determines the number of connections within the subcells, and external sparsity determines the number of connections between subcells (i.e., between input streams). Each subcell (A_1 to B_2) is given a unique combination of seeds for generating the connection matrices; [1, 2, 3, 4] for A_1, [5, 6, 7, 8] for A_2, etc. This resulted in a total of 576 configurations.

Results

The overall success rate of all logic gates can be found in figure 14. The results per logic gate for this version as compared to the previous version are displayed in table 7. The results of the 'Control nodes' version are taken from experiment 3, table 6. Splitting of input streams did not result in any complete successes for any configuration. Rather, there is a small increase in failures as opposed to partial successes.

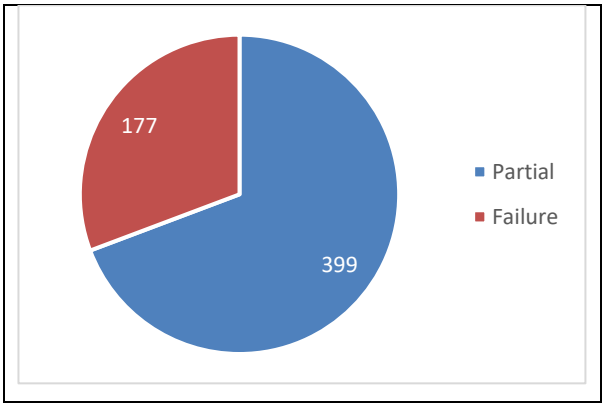


Figure 14: Success ratio of all logic gates using control nodes for learning and splitting input streams.

Table 7: Success rate per logic gate for learning with separate input streams vs control nodes with subcells.

Logic Gate	***Split input***		***Control nodes***	
OR	*399*	*(69,3%)*	*404*	*(70,1%)*
NOR	*397*	*(68,9%)*	*403*	*(70,0%)*
AND	*178*	*(30,9%)*	*242*	*(42,0%)*
NAND	*179*	*(31,1%)*	*245*	*(42,5%)*
XOR	*2*	*(0,3%)*	*23*	*(4,0%)*
XNOR	*24*	*(4,2%)*	*16*	*(2,8%)*
Inter. XOR	*86*	*(14,9%)*	-	
Inter. XNOR	*77*	*(13,4%)*	-	

Generally, the splitting of input streams appears to hinder convergence. The success rate of every logic gate drops, with the exception of XNOR. For OR / NOR, the decrease in success rate is insignificant (<2%), but for AND / NAND, there is a considerable difference. Interestingly, the XNOR is solved with more configurations, but the success rate of XOR dropped to nearly 0. Appendix A provides more information on which parameter settings affected the success ratio for all linear gates and individually for XOR / XNOR.

A closer inspection of the output activations reveals that there are configurations for which the 'intermediate' gates both solved (i.e., OR / NAND for XOR, NOR / AND for XNOR, see figure 5c), but their summed outputs were not suitable for solving the overall gate. These numbers are presented in table 7 as "Inter XOR" and "Inter XNOR", respectively. These configurations had the potential to solve, but adding up the output values for each subcell pair ($A_1 + A_2$ and $B_1 + B_2$) was not sufficient to solve XOR or XNOR. It is possible that the output activations for one subcell pair are much higher than those of the other. In this case, the activation of one solved logic gate will have only a small bearing on the summed outputs. The 'input' C5 has to work with is then skewed and does not match the situation as shown in figure 6c, meaning the overall gate cannot solve.

9. Experiment 5:

Saturation cells

In order to even out the output levels of both cells, we introduced saturation cells between the pairs A and B and the overall output layer (see figure 15). Saturation cells are clusters (reservoirs) of densely interconnected nodes, that drive positive activation they receive as input to a maximum and negative activation to zero. This way, the output levels will more

closely resemble the tables shown in figure 5c. Having similar activation levels from both subcells, it should be easier for the overall cell to converge for the non-linear gates, because there will be no more cases in which one subcell pair has such a strong activation level that the other subcell pair cannot compensate for.

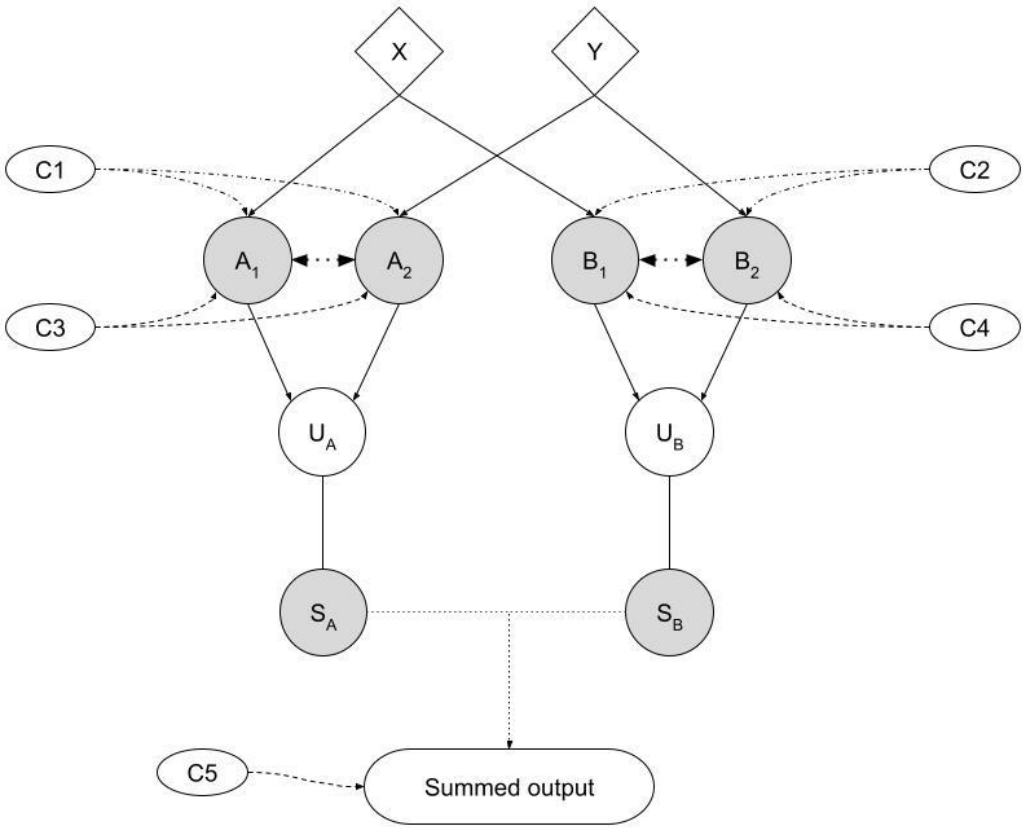


Figure 15: Depiction of the simulation of the dopant cell reservoir with control nodes and saturation cells. Input X is given to A_1 and B_1, and Y to A_2 and B_2. Saturation cells s_A and s_B saturate the activation received from subcells [A_1, A_2] and [B_1, B_2], respectively. All control nodes still behave in the same manner.

The parameters used for generating the subcells are the same as in the experiment 4. However, due to time constraints, we decided to only test configurations with the most successful values for cell size; 20 and 50 (see the analysis of the results of experiments 1-4 in appendix A). This resulted in a total of 288 iterations. The saturation cells were both created using a cell size of 20, an internal sparsity of 0 (meaning every node is connected to every other node), seeds [0, 0, 0, 0] and a connection strength of 0.01 as to not inflate the activation levels to a point where C5 cannot reduce it within the predetermined 1000 learning runs.

Results

The overall success rate can be seen in figure 16. Table 8 shows the success rate per logic gate, contrasting the current version (saturation) to the previous version (split input streams

without saturation; taken from table 7). Given the fact we limited the cell size to the two most successful settings, it is reasonable to assume that the percentage of success would go up for every logic gate as compared to previous versions.

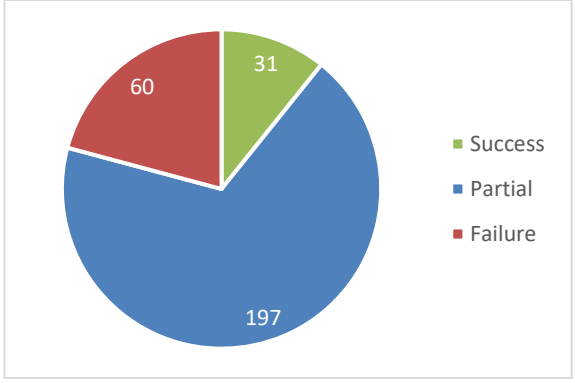


Figure 16: Success ratio of all logic gates using control nodes for learning and saturation cells.

Table 8: Success rate per logic gate for learning with saturation cells vs control nodes without saturation.

Logic Gate	***Saturation***		***No saturation***	
OR	*228*	*(79,2%)*	*399*	*(69,3%)*
NOR	*205*	*(71,2%)*	*397*	*(68,9%)*
AND	*103*	*(35,8%)*	*178*	*(30,9%)*
NAND	*105*	*(36,5%)*	*179*	*(31,1%)*
XOR	*35*	*(12,2%)*	*2*	*(0,3%)*
XNOR	*45*	*(15,6%)*	*24*	*(4,2%)*

Unlike the previous versions, there are now configurations in which all logic gates converge (around 10,7% of all settings). This shows the saturation effect is beneficial for solving all logic gates together with a single configuration. The percentage of complete failures has dropped, whereas the partial success rate is approximately equal. Some configurations that were partial successes in the previous version now enable the XOR and XNOR to solve as well, resulting in full successes. The percentage of partial successes remains equal, because there are less 'unfit' configurations due to the limitations on the cell size.

The increase in success rate for OR / NOR can be attributed to the same fact. There are less configurations that did not allow for pathways to form between input and output nodes, because these existed mostly when the cell was too small (i.e., cell size 10). Taking out these configurations increases the percentage of successes, but not the capabilities of the configurations that lead to failures in previous versions.

AND / NAND still have a lower success rate compared to the control nodes without splitting input pathways (experiment 3). Although the success rate is a bit higher as compared to the previous version, it is not significant enough to attribute it to the saturation cells, but more likely due to the filtering of cell sizes.

For both XOR and XNOR, the saturation cells are a beneficial addition. When the output levels of both subcells are driven to a maximum, the summed outputs can be adjusted in such a way that XOR / XNOR solve (as in figure 6c). The saturation cells make this

possible. The relatively low success rate could be caused by subcell pairs not solving, and therefore driving the wrong activation to a maximum. Those summed activations can then not be adjusted by just C5, making the overall gate unsolvable for this setup.

10. Experiment 6: Semantic network

The last experiment entails implementing part of the Parallel Distributed Processing (PDP) model of semantic relations by McClelland and Rogers (2003), using one or multiple reservoirs and control nodes for learning. In order to maximize the odds of success, it is important to choose the most suitable structure given the results of the previous experiments, meaning the structure that is best capable of solving the logic gates. Overall, the version that is most suited for solving all logic gates is the structure presented in experiment 5 (saturation cells). This version shows the highest overall success rate, and the highest success rates for both non-linear gates. However, when only looking at the success rates for linear gates, the first structure that introduced control nodes (i.e., experiment 3) exceeds the saturation cell structure (see also appendix A for the results of linear gates).

As described in section 2, the PDP model contains items, relations and attributes. It is capable of finishing three-item propositions, meaning each item-relation pair should result in the correct attributes. For this purpose, the item and relation are considered the input, and the correct attributes are the output of the model. This can be realized by using the AND gate, as the model should only return attributes that are the correct output for both the item and relation. The AND gate is capable of differentiating attributes that are linked to both the item and relation to those that are linked to only one or neither.

The current program works by giving all combinations of input (i.e., every item-relation pair) to each cell (i.e., attribute). For testing purposes, we only modeled the 'animal' branch of the semantic network (see figure 15). Therefore, it contains four items (*'robin', 'canary', 'sunfish'* and *'salmon'*), four relations (*'has', 'can', 'is'* and *'isa'*), and a total of 14 attributes (*'animal', 'bird', 'fish', 'move', 'skin', 'feathers', 'fly', 'wings', 'red', 'sing', 'yellow', 'gills'* and *'scales'*).

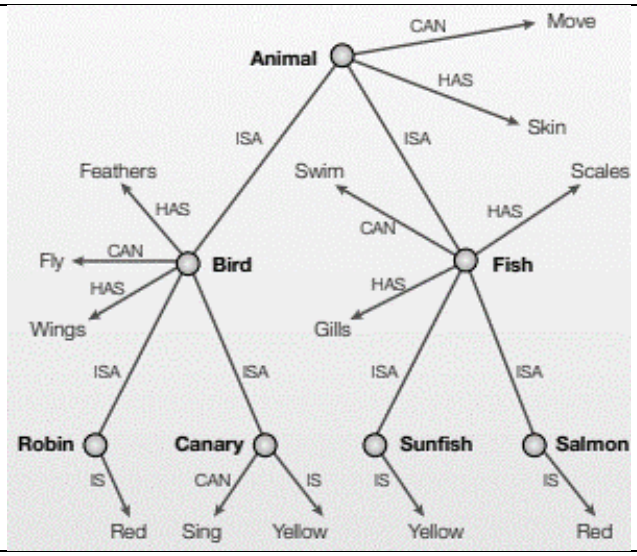
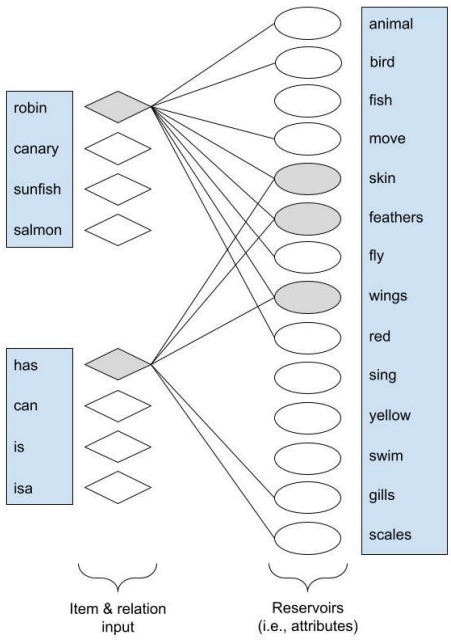


Figure 15: Animal branch of the semantic network used by McClelland and Rogers (2003) in their Parallel Distributed Processing (PDP) model of semantic relations. Figure taken from McClelland and Rogers (2003) and adapted to show only the animal branch.

Each reservoir individually represents an attribute and attempts to solve the AND gate on the 16 input patterns (one per item-relation pair; see figure 16a). The output pattern of each cell is then an array of 16 numbers, where a positive number means this attribute is true for the given proposition. To exemplify, the attribute *'feathers'* receives an input of 1 for items *'robin'* and *'canary'*, and for the relation *'has'*. All other items and relations give an input of 0. Solving the AND gate for these inputs should therefore only result in a positive outcome for the combinations *'robin has'* and *'canary has'*. This process is the same for every attribute, resulting in 14 arrays of 16 output values (see figure 16b for example output arrays for *'animal', 'feathers'* and *'swim'*).

Getting the output per item-relation pair rather than per attribute is done by taking the output values corresponding to the item-relation pair from each of the 14 output arrays (e.g., taking the output values for input number 1 for *'robin has'*; see figure 16b). The position of the output value for each input pattern is consistent for all of these arrays. If every attribute has been solved correctly, it should follow that every item-relation pair results in the correct output. Therefore, when taking the positive values for *robin has*, the resulting attributes should be *['feathers', 'wings']*.



a

Input number	Item-relation	Output 'animal'	Output 'feathers'	Output 'swim'
1	robin has	0	1	0
2	robin can	0	0	0
3	robin is	0	0	0
4	robin isa	1	0	0
5	canary has	0	1	0
6	canary can	0	0	0
7	canary is	0	0	0
8	canary isa	1	0	0
9	sunfish has	0	0	0
10	sunfish can	0	0	1
11	sunfish is	0	0	0
12	sunfish isa	1	0	0
13	salmon has	0	0	0
14	salmon can	0	0	1
15	salmon is	0	0	0
16	salmon isa	1	0	0

b

Figure 16a: Schematic representation of the current implementation of the Parallel Distributed Processing (PDP) model of semantic relations by McClelland and Rogers (2003). Each attribute is represented by a single reservoir that attempts to solve the AND gate for all 16 input patterns (item-relation pair). Pictured is the input for 'robin has'. *Every input node is connected to every reservoir, but only gives a positive input to attributes that are true for the specific item or relation. Pictured are the connections from each input node to the attributes that are true for* 'robin' *and* 'has'. *The highlighted attributes are the correct output attributes for this item-relation pair.*
b: Examples of (desired) output arrays per attribute. Seen here are the attributes 'animal', 'feathers' *and* 'swim'. *Every attribute returns an array of size 16, resulting in 14 output arrays. Output per item-relation pair (i.e., command) is returned by fetching the corresponding value from each of the 14 output arrays.*

As the PDP model only makes use of the AND gate for solving the three-item propositions, the highest success rate is to be expected by implementing the structure of the reservoir that is most suitable for solving this logic gate. Therefore, the structure that is applied here is similar to the one in experiment 3, meaning each attribute is represented by a single reservoir. However, in the current version these reservoirs are not interconnected. Connecting certain reservoirs together would mean these would influence each other's activation levels, meaning they would result in the same activation state (i.e., positive or negative) more often, suggesting these attributes should be seen as a group.

Each reservoir has its own set of control nodes that work similarly to previous iterations. One type of control nodes is used for reversing input (C1/C2), one for adding or subtracting activation in the reservoir (C3/C4), and one for adding or subtracting activation from the summed output (C5/C6). A schematic representation of the implemented structure can be found in figure 17.

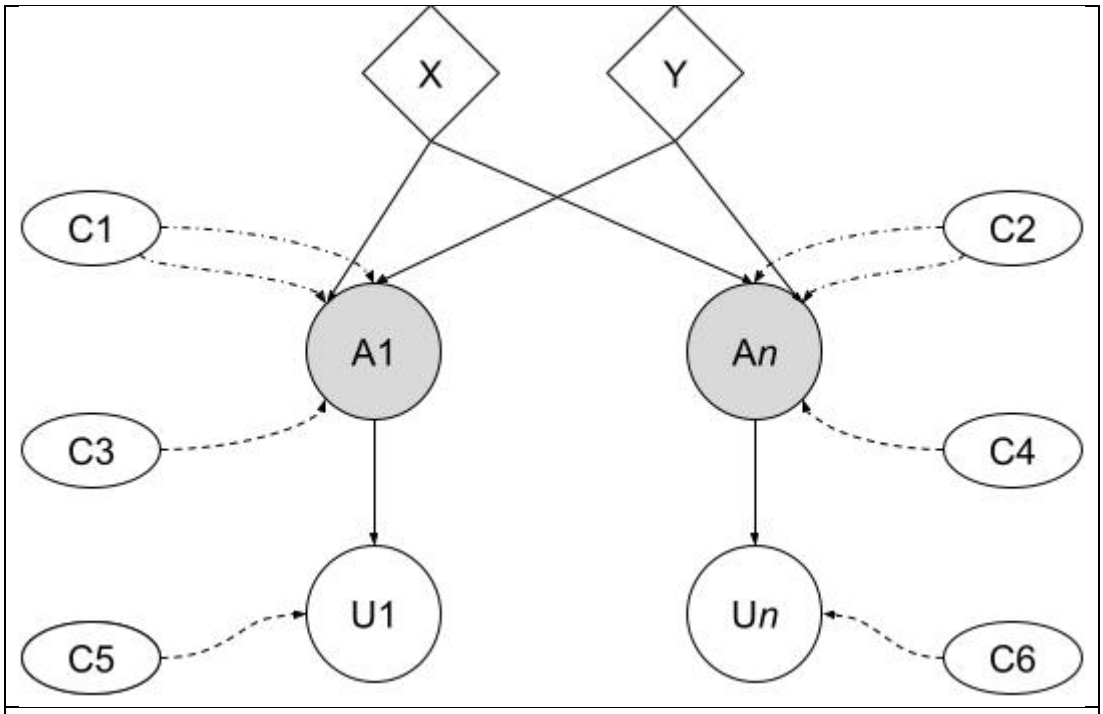


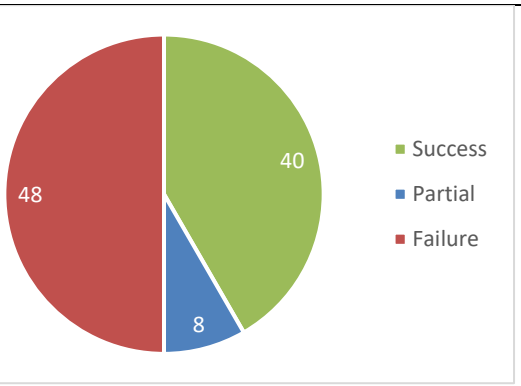
Figure 17: Schematic representation of the reservoirs and control nodes for learning in the current implementation of the PDP model. Pictured are two reservoirs. In total, 14 reservoirs are used to represent all attributes in the animal branch of the semantic network. Each reservoir has its own set of control nodes, but receive the same input patterns (X for item, Y for relation).

Previous experiments have shown that the reservoir is best capable of solving logic gates with certain configurations, such as cell sizes 20 and 50, a higher sparsity setting and low connection strength. The seed appeared to have a minimal effect on the success ratio (see appendix A). However, the differences were usually not so great as to justify not testing further iterations with the full range of settings used in previous experiments, especially considering the limit on the number of configurations due to the absence of external sparsity. Therefore, the semantic network has been tested with all settings for cell size (10, 20, 50 and 100), internal sparsity (0.80, 0.85, 0.88, 0.92, 0.95 and 0.98) and connection strength (0.2, 0.5, 0.8 and random).

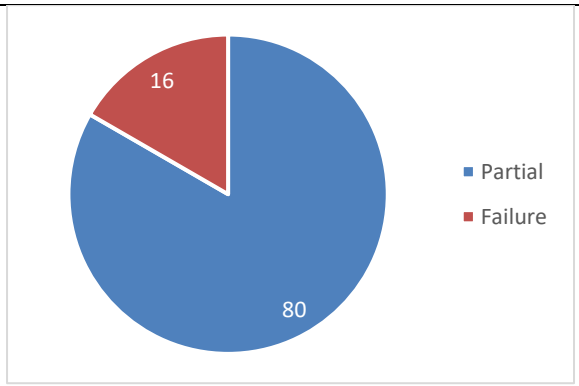
For the seed parameter, we initially tested the same setting for each reservoir (namely [1, 2, 3] for input to reservoir, within reservoir and reservoir to output, respectively). Given the difficulty of replicating the exact same physical reservoir, we further tested the semantic network with a unique seed combination for each reservoir (i.e., [1, 2, 3] for the first attribute, [4, 5, 6] for the second, etc.). Using these settings, the program ran a total of 96 configurations for the 'static' seed setting, and the same 96 configurations using unique seeds for each attribute.

Results

The overall success rate for the single seed configurations can be found in figure 18a, and for the unique seeds per cell in 18b. Here, "success" means every cell (i.e., attribute) gave the correct output for every input combination (i.e., item-relation pair) using the same configuration of parameters. "Partial" means some, but not all attributes solved, "failure" means none of the attributes gave the correct output pattern.



a



b

Figure 18: Success ratio of all attributes for every input combination using control nodes for learning. a: Success ratio for single seed configurations (i.e., using the same seed combination for every reservoir). b: Success ratio for unique seed configurations.

For the single seed configurations, there are configurations that lead to full successes and others that lead to partial successes. Interestingly, the successes and partial successes together amount for exactly half of the configurations. The success rate matches the results of the AND gate in experiment 3, on which the current structure was based. For the unique seed configurations, there were no configurations for which every reservoir was able to reach convergence. Appendix A details which configurations resulted in successes, partial successes and failures for both seed settings.

Partial successes for the single seed condition seem counterintuitive, as the pathways in all reservoirs are the same, suggesting that either all cells should be capable of solving the AND gate (thus leading to full successes) or none should be able to (leading to failure). However, a closer inspection of the initial untrained output values reveals how partial successes exist. It appears that the pathways from the 'relation' input node to the output node have a much bigger impact on the total activation than the pathways from the 'item' input node. This results in similar activation levels for input [1,1] (i.e., correct item and correct relation for any given attribute) and [0,1] (i.e., incorrect item and correct relation). The difference between [1,1] and [1,0] (i.e., correct item and incorrect relation) is large enough to differentiate. This makes it possible to solve for attributes that are true for all items, but not

attributes that are true for only some of the items (see table 9). Therefore, the attributes *animal*, *move* and *skin* are able to solve, as these are true for all four items (with different relation inputs).

Examining the full successes further illustrates the ease of solving the attributes that are true for all items as compared to the others. The attributes that are true for all four items often solved in less learning runs as compared to those that only applied to some items. However, this pattern is less pronounced in the cell size 100 condition (see table A37 in Appendix A). It appears that ensuring there are many connections will even out the effect on the total activation by each input node. Additionally, using a different seed might also remedy these problems, as this will allow for different pathways that could balance the effect of each input node.

Table 9: Success rate per attribute (i.e., cell) in the semantic network using the same seed for all reservoirs

Attribute	***Success rate***	
animal	*48*	*(50,0%)*
bird	*40*	*(41,7%)*
fish	*40*	*(41,7%)*
move	*48*	*(50,0%)*
skin	*48*	*(50,0%)*
feathers	*40*	*(41,7%)*
fly	*40*	*(41,7%)*
wings	*40*	*(41,7%)*
red	*40*	*(41,7%)*
sing	*40*	*(41,7%)*
yellow	*40*	*(41,7%)*
swim	*40*	*(41,7%)*
gills	*40*	*(41,7%)*
scales	*40*	*(41,7%)*

Table 10: Success rate per command (i.e., input combination) in the semantic network using a unique seed combination per reservoir

Command	***Success rate***	
robin has	*40*	*(41,7%)*
robin can	*40*	*(41,7%)*
robin is	*40*	*(41,7%)*
robin isa	*40*	*(41,7%)*
canary has	*40*	*(41,7%)*
canary can	*40*	*(41,7%)*
canary is	*40*	*(41,7%)*
canary isa	*40*	*(41,7%)*
sunfish has	*40*	*(41,7%)*
sunfish can	*40*	*(41,7%)*
sunfish is	*40*	*(41,7%)*
sunfish isa	*40*	*(41,7%)*
salmon has	*40*	*(41,7%)*
salmon can	*40*	*(41,7%)*
salmon is	*40*	*(41,7%)*
salmon isa	*40*	*(41,7%)*

There were no differences in the success rates per command (i.e., input combination; see table 10). This result is unsurprising, as it matches the number of total successes. When each attribute is solved by a certain configuration, it also means that every three-item proposition is finished correctly by the system. However, there are no three-item propositions that can correctly be completed using only *'animal', 'move'* and *'skin'*, meaning partial successes do not increase the success rate per command.

The success rate per cell (or attribute) for the unique seed condition can be found in table 11. Although there are no full successes, each attribute is capable of solving for certain

configurations with a variable success rate (from 29,2 up to 69,8%; average 43,7%). This variability is likely caused by the different seeds creating different pathways between input and output nodes. There is no apparent structure in the success rates per attribute. Given the results of the single seed configurations, it could be expected that the attributes *move*, *grow* and *skin* have a higher success rate, but the different pathways in each reservoir ensure that the contribution of the 'relation' input is not overbearing the 'item' input.

The success rate per input combination also shows varying results (see table 12), but the differences are not as big as the success rate per attribute (ranging from 15,6% to 31,3%; average 21,4%). Furthermore, the highest success rate per input combination is less than a third of all tested configurations. There is no discernable pattern in the faulty output either; in some cases, there are too many attributes returned, and some cases in which there are too few attributes. Similar to the success rate per attribute, there does not seem to be a pattern in which commands give similar output levels.

Table 11: Success rate per attribute (i.e., cell) in the semantic network using a unique seed combination per reservoir

Attribute	***Success rate***	
animal	*56*	*(58,3%)*
bird	*30*	*(31,3%)*
fish	*30*	*(31,3%)*
move	*44*	*(45,8%)*
skin	*56*	*(58,3%)*
feathers	*43*	*(44,8%)*
fly	*37*	*(38,5%)*
wings	*67*	*(69,8%)*
red	*41*	*(42,7%)*
sing	*41*	*(42,7%)*
yellow	*29*	*(30,2%)*
swim	*52*	*(54,2%)*
gills	*33*	*(34,4%)*
scales	*28*	*(29,2%)*

Table 12: Success rate per command (i.e., input combination) in the semantic network using a unique seed combination per reservoir

Command	***Success rate***	
robin has	*23*	*(24,0%)*
robin can	*30*	*(31,3%)*
robin is	*24*	*(25,0%)*
robin isa	*16*	*(16,7%)*
canary has	*20*	*(20,8%)*
canary can	*24*	*(25,0%)*
canary is	*24*	*(25,0%)*
canary isa	*15*	*(15,6%)*
sunfish has	*17*	*(17,7%)*
sunfish can	*24*	*(25,0%)*
sunfish is	*20*	*(20,8%)*
sunfish isa	*17*	*(17,7%)*
salmon has	*17*	*(17,7%)*
salmon can	*23*	*(24,0%)*
salmon is	*18*	*(18,8%)*
salmon isa	*16*	*(16,7%)*

These results suggest that the current setup using unique reservoirs for each attribute is not optimal for running a semantic network, as it is already incapable of solving a small number of attributes. Increasing the number of attributes and items would likely only lead to more partial successes, because there is more variation in the cells. In order to make the semantic network functional, every reservoir needs to have a sufficient number of connections, as well as equal contribution from both input nodes. Simply increasing the

number of nodes in each reservoir could balance out the effect of both input nodes, but this solution may not be sufficient, given that cell size 100 was successful in the single seed configurations, but not in the unique seed ones (see appendix A).

As more items and attributes are added to the network, more reservoirs are needed that contain pathways that allow for equal contribution of both input nodes. This could be a problem given the lack of reproducibility of the cells, as it is impossible to run a semantic network using the same cell for each attribute. However, the second part of this experiment shows that the reservoirs in the network do not need to be exact copies, as there were configurations for every individual attribute to solve. Nevertheless, it could require testing many different setups to ensure equal contribution from both input nodes.

Another approach could be to split the input streams and reintroduce saturation cells between the reservoirs of each input stream and the output node in order to balance out the activation from each input node. This also invites more room for error, as it would increase the number of reservoirs and therefore add more variability.

11. General discussion

The goals of the current research were to simulate the structure of the boron dopant cell as developed by Chen et al (2020) and to determine its capabilities of running a small-scale semantic network. We simulated the structure of the cell with a reservoir containing randomly interconnected nodes, that is capable of transferring an input 'current' from the input nodes to the output node with five control nodes to program the output activation. Using this reservoir, we replicated part of Chen et al's (2020) research in programming linear and non-linear logic gates, and expanded on their research by implementing the Parallel Distributed Processing model of semantic relations (McClelland & Rogers, 2003).

The first two experiments were meant to test the capabilities of our reservoir with a known functioning learning algorithm called perceptron learning. The most basic perceptron should have a 100% success rate for linear gates, but is not suitable for non-linear logic gates without an additional layer of nodes. Our results show that a single reservoir containing *n* nodes adds a complexity that reduces the success rate of perceptron learning for linear gates, but which makes it possible for the non-linear gates to solve with certain configurations of the cell. Adding another reservoir to simulate a perceptron with a hidden layer further increases

this complexity, making it more suitable for solving the OR / NOR gates, but less suitable for all other gates.

In experiments 3 and 4, we implemented a set of control nodes to program the output instead of using the perceptron learning algorithm. The first set of control nodes inverses the input connections, the second set adds to or subtracts from the activation within a reservoir, and the last control node adds or removes activation from the overall cell output. Using two interconnected reservoirs to solve separate logic gates (like in experiment 2), the results show that control nodes are a better way of programming the output, as it leads to a higher success ratio for all logic gates. Splitting the input streams to make sure the intermediate logic gates do not influence each other's activation leads to less successes, mainly because of a discrepancy between the output activations of each subcell pair.

These first experiments provided a good baseline that translates well to how the boron dopant cell works. Much like the physical cell, the reservoirs were capable of transferring an input signal (i.e., conducting a current) from input nodes through the cell to an output node. Furthermore, these experiments showed that it is possible to program the output activation using control nodes with basic functionality of reversing input activation, and adding or subtracting activation inside the cell. Although there were no configurations with full successes in these experiments, these aspects are still promising. The physical cell is much bigger in size and therefore contains many more potential pathways, and the control nodes could be utilized to more precisely program the output activation.

The experiments also illustrated the difficulty of the non-linear logic gates as compared to their linear counterparts. The XOR / XNOR gates showed a much lower success rate in all four experiments. In the research by Chen et al. (2020), it was already apparent that these gates were harder to solve. The output activations for XOR / XNOR were very different from the other gates (approximately 0.04V for XOR, 0.1V for XNOR versus 0.2 for AND, NAND and OR, and 0.4V for NOR). Therefore, the lower success rate of the non-linear gates found in the current research were in line with expectations.

Furthermore, the first experiments showed that some configurations are more successful than others for certain logic gates (see also appendix A). Every parameter used to generate the reservoirs had a significant impact on the success ratio, with the only exception of the seed combination. This suggests that the number of potential pathways is critical for successfully solving logic gates using one or multiple reservoirs. Having too few connections

could mean there is no activation flow from the input nodes to the output node, whereas having too many connections could lead to an overflow of activation when any input is given.

Given that performance relies on the number of pathways and to some degree which pathways between input and output nodes, replicability plays a big part in the potential applications of the physical boron dopant cell. In the simulations, it was possible to recreate a reservoir with the exact same structure as one that was utilized previously. For a physical cell, it is impossible to dope the boron onto the silicon in the exact same manner twice, meaning each cell is unique. Therefore, it is possible there are great differences in performance between any two cells. The physical cell contains many more nodes and potential connections, meaning the exact configurations of the pathways is less impactful, but the difference in output levels suggest that enough pathways are a required but not sufficient factor for solving all logic gates.

In experiment 5, we tried to counter these problems by introducing saturation cells (i.e., clusters of densely interconnected nodes) between the reservoirs and output layer. These saturation cells standardize the output by driving the activation levels to a local maximum. Then, the last control node is utilized to solve the logic gate. The results show that this system is capable of solving all logic gates under certain configurations, and that this structure is the most suited for solving XOR / XNOR.

Although the use of saturation cells was not required for the physical cells to solve the non-linear logic gates, applying some saturation effect on the activation levels would be beneficial when the output of one cell is used as input for another. As the output levels for XOR and XNOR differ greatly from the output of the linear logic gates (Chen et al., 2020), these activation levels could not be used in one system as the activation of XOR / XNOR would hardly impact the total activation in another cell when the other input comes from a linear gate. The use of saturation would make it possible to chain multiple boron dopant cells and could therefore be a useful contribution to the overall system.

In the final experiment, we trained a network of reservoirs to finish three-item-propositions, like the Parallel Distributed Processing (PDP) model of semantic relations by McClelland and Rogers (2003). This version consisted of multiple reservoirs, that each represented one attribute. Every reservoir received the same input sequence, namely every combination of item and relation, and each reservoir individually aimed at solving the AND gate on this given sequence. For testing purposes, only the ‘animal’ branch of the semantic

network previously implemented by McClelland and Rogers (2003) was programmed into the current network, resulting in a network consisting of 14 reservoirs and 16 input signals. The output values for every input signal are stored in an array (one for each reservoir), and the output for each item-relation pair is generated by taking the corresponding values from each of the 14 output arrays (e.g., for the item-relation pair *'robin has',* the corresponding output values are the first number of every array).

In this experiment, we tested two different setups. In the first setup, each reservoir was generated using the same seeds, resulting in identical reservoirs. In the second setup, each reservoir was given a unique combination of seeds, resulting in unique reservoirs for each attribute. When using the same seed setting for each reservoir, there were configurations that were capable of solving every attribute, but there were also some in which only attributes that applied to all items solved. This had to do with the impact each input pathway had on the output activation. When using a unique seed for each reservoir (i.e., when each reservoir contains different pathways between input and output nodes), there were no configurations that resulted in a successful outcome for all attributes. However, each attribute was solvable by some configurations, which could suggest that full successes are possible for a physical cell that contains more nodes and therefore more potential connections.

The last experiment gave some insights into the requirements for adapting the boron dopant cell to implement a semantic network. It showed that a semantic network is easiest to run using identical cells. This is a problem for the physical boron cell, as it is hard to reproduce the same cell due to the way the boron atoms are placed on the silicon. However, the experiment also showed that identical cells are not required, as each unique seed combination allowed for the attribute to solve given the right configuration. Complete failures may be avoided by having the right number of connections, as in the previous experiments, whereas partial failures could be prevented when both input streams have an equal effect on the total activation of the cell. This suggests that using multiple reservoirs is a viable approach, but it may take time to find enough correct configurations, especially as the size of the network increases. Alternatively, the use of saturation could prove beneficial in this area.

11.1 Limitations of the study

The current approach of simulating the boron dopant cell has some limitations. Firstly, the configurations we could test were severely limited due to computational cost and time constraints. This only allowed us to test cells up to a maximum size of 100 nodes, whereas a physical dopant cell might contain millions of atoms. Furthermore, it is possible that the

physical cell contains a multitude of ‘reservoirs’ (i.e., more heavily interconnected clusters of nodes), while we only tested up to six reservoirs when including saturation. Therefore, it is possible that we found successful configurations that are not replicable with a physical boron dopant cell, or that the configurations required to make the physical cell fit to solve the logic gates were not discovered in the current research.

Secondly, the reservoir does not simulate the quantum-mechanical behavior of the cell, partly due to our limited understanding of it. In our simulations, we used a reservoir in combination with a squashing function in place of the hopping regime described by Chen et al. (2020), under the assumption that it would spread activation through the cell in a similar manner. This approach does not take the physical aspects of the cell into consideration, but the overall functioning of the cell (i.e., its ability to conduct a current from the input nodes through the cell to the output node) and its structure is maintained. While there may be options to mimic the quantum-mechanical behavior of the cell, we opted not to pursue these, in first place because we lack the computational power to run these programs and in second place because the physical properties of the cell are not crucial for the goals of the current research.

Thirdly, the functionality of the control nodes in the simulation was assumed to be simple. It is possible that the control nodes designed by Chen et al (2020) influence the activation within the cell in a different, more complex manner, as it is stated the control nodes affect the “potential landscape of the cell”. Furthermore, in a physical system, it is likely that control nodes mostly influence atoms in close proximity to the control input. In the current research, this is simulated by having the control nodes influence specific reservoirs, but this might not have the same effect.

Lastly, in order to solve the XOR / XNOR logic gates, we had to introduce saturation cells to standardize the output activation from each ‘intermediate’ reservoir. Even when using saturation cells, the success rate of the XOR / XNOR gates are relatively low. This could suggest that the current setup is not as suitable as the physical cell created by Chen et al. (2020). However, as stated before, the output activation for XOR / XNOR of the physical cell is completely different than that of all linear logic gates. This shows that these logic gates are challenging to solve with the physical system as well. Furthermore, if the physical cell would be used in a system combining multiple of these cells, there would also be a need to standardize the output levels, as the output of XOR / XNOR would be insignificant as compared to other output activations and therefore not usable as input for another layer.

Although these limitations may mean that our simulation is not a fully accurate representation of the boron dopant cell, the overall functionality and structure does match; a cell that is capable of conducting a current from two input nodes through a 'hidden layer' to one output node, using five control nodes to program the output. In other words, on the surface level, the simulation works similar enough to the dopant cell. Therefore, it is still a good indication of what one or more boron dopant cells could be capable of in terms of logic gate solving and semantic networks. Furthermore, suggested use of the control nodes in the current research could provide a viable, easy to implement alternative to perceptron learning. Lastly, the addition of saturation cells was beneficial in solving the non-linear gates in our research and could prove to be a proper solution to the problematic output activation levels of the physical cells.

11.2 Suggestions for future research

In order to fully understand what the boron dopant cell can do for semantic networks, further testing is required. In first place, the current structure needs to be extended and tested on a system with more computational power. This way, it is possible to increase the number of nodes and test how this influences the capabilities of the cell. Optimally, the semantic network should be programmed using a physical boron dopant cell. Alternatively, more simulation studies could provide further insight.

Before actually applying the boron dopant cell, it should be investigated how multiple cells interact in a bigger system. Hirjibehedin (2020) claims the cell developed by Chen and colleagues (2020) is "inherently scalable" and that "individual classifier devices can be run in parallel without any conflicts". However, as evidenced by our simulations and the output levels of the dopant cell, the XOR and XNOR gates are much more difficult to solve than the linear gates and the output activation is entirely different (see figure 3c, from Chen et al., 2020). This suggests that the boron dopant cell needs some method of standardizing the output before it can be utilized in a bigger system. This would also increase the performance of a semantic network, as all input values should impact the output activation in a similar manner.

Furthermore, the reproducibility of the boron dopant cell could be an issue. As the current results have shown, the configuration of the cell impacts the capabilities for it to solve the logic gates and, by extension, the correct output for item-relation pairs in a semantic network. However, for the semantic network, using the same seed setting for all cells (i.e., generating two identical cells) greatly benefits the performance of the network. Hirjibehedin

(2020) already states that it is difficult to reproduce the same physical cell, meaning the performance of two of these boron dopant cells could vary greatly. Therefore, extensive testing of configurations is required to find out which settings are optimal for running a small semantic network, before it can be scaled up to a human-like knowledge base.

Aside from the research on the physical cell, there are also topics of research with regards to the implementation of the cell. In the current study, the desired output patterns for all attributes in the semantic network are hard-coded. The original PDP model makes generalization between similar concepts possible due to the similarity in connection weights for these items. When new information enters the system, it only needs to adjust certain connection weights to differentiate the new concept from pre-existing ones (McClelland & Rogers, 2003; Rogers & McClelland, 2008). As the current model does not allow for changing of connection weights, it should be researched how this structure can still account for the general-to-specific acquisition of information phenomena using control nodes instead.

Another approach that could be considered is utilizing the boron dopant cell (or a similar system) to other steps of the Natural Language Analysis process. The energy-efficiency could prove beneficial for the lower-end processes of NLA as well, such as Part-of-Speech tagging. Östling (2018) tested neural network methods for Part-of-Speech tagging and compared the accuracy and efficiency to traditional perceptron-based systems. The older systems outperformed the neural network methods both on accuracy and efficiency (Östling, 2018). However, it is worth investigating whether using the boron dopant cell is more energy-efficient due to its structure.

12. Conclusion

In the current research, we have attempted to replicate and extend the research on a boron dopant cell by Chen et al. (2020) by means of simulating the physical cell as a reservoir. The goal of the research was to determine the capabilities of a system like the boron dopant cell in terms of running a small-scale semantic network.

In the first four experiments, we simulated the functionality and structure of the boron dopant cell by (1) introducing a reservoir to transfer an input signal through a hidden layer to the output node; (2) adding another reservoir to add ‘intermediate’ logic gates; (3) implementing simple control nodes; and (4) splitting the input streams to remove connectivity

between intermediate logic gates and to balance the effects of both input nodes on the output activation.

The results showed that adding reservoirs introduces complexity and therefore reduces the success rate of linear logic gates as compared to a basic perceptron, but also allows for non-linear gates to solve under certain configurations. Using control nodes proved to be a more effective learning algorithm when using one or more reservoirs. However, the success rate of the non-linear gates was still low, mostly due to the unequal contribution of the intermediate logic gates.

In the fifth experiment, we attempted to solve this problem by introducing saturation to standardize the output of the intermediate gates. This allowed for configurations that solved all logic gates, but the success rates for the non-linear logic gates were still low. These results suggest that the reservoir only works under certain configurations and that saturation is a required, but not sufficient, method for converging on non-linear logic gates.

In the final experiment, we used multiple reservoirs to implement part of the Parallel Distributed Processing model by McClelland and Rogers (2003). The system was able to finish three-item propositions, but full successes were only present when every attribute was represented by identical reservoirs. When each reservoir had unique pathways between the input and output nodes, there was too much variability in the correct configuration setting for each reservoir, leading to varying success rates per attribute. Given the difficulties of reproducibility addressed by Hirjibehedin (2020), this could be problematic for the physical boron cell.

Given the limitations on the number of configurations run in the current experiment, it is not unthinkable that configurations of the physical cell exist for which a small semantic network is capable of giving the correct output for all attributes. Scaling the network up is going to be yet another challenge, as each attribute added to the network will increase the complexity and variability, leaving more room for error. In short, the boron dopant cell might prove to be effective for running a semantic network, but a lot of research is required to find the right configurations.

References

Brown, R. E. (2020). Donald O. Hebb and the Organization of Behavior: 17 years in the writing. *Molecular Brain*, *13*(1), 1–28. https://doi.org/10.1186/s13041-020-00567-8

Chen, T., van Gelder, J., van de Ven, B., Amitonov, S. V., de Wilde, B., Ruiz Euler, H. C., Broersma, H., Bobbert, P. A., Zwanenburg, F. A., & van der Wiel, W. G. (2020). Classification with a disordered dopant-atom network in silicon. *Nature*, *577*(7790), 341–345. https://doi.org/10.1038/s41586-019-1901-0

Chowdhary, K. R. (2020). Fundamentals of artificial intelligence. In *Fundamentals of Artificial Intelligence*. https://doi.org/10.1007/978-81-322-3972-7

Collins, A. M., & Loftus, E. F. (1975). A spreading-activation theory of semantic processing. *Psychological Review*, *82*(6), 407–428. https://doi.org/10.1037/0033-295X.82.6.407

Deliyanni, A., & Kowalski, R. A. (1979). Logic and Semantic Networks. *Communications of the ACM*, *22*(3), 184–192. https://doi.org/10.1145/359080.359090

Hausser, R. (2001). Foundations of Computational Linguistics. In *Foundations of Computational Linguistics*. https://doi.org/10.1007/978-3-662-04337-0

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). https://doi.org/10.1371/journal.pone.0052946

Hirjibehedin, C. F. (2020). Evolution of circuits for machine learning. *Nature*, *577*(7790), 320–321. https://doi.org/10.1038/d41586-020-00002-x

Jaeger, H., & Haas, H. (2004). Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. *Science*, *304*(5667), 78–80. https://doi.org/10.1126/science.1091277

Khurana, D., Koli, A., Khatter, K., & Singh, S. (2017). *Natural Language Processing: State of The Art, Current Trends and Challenges*. http://arxiv.org/abs/1708.05148

Lehmann, F. (1992). Semantic Networks. *Computers & Mathematics with Applications*, *23*(2–5), 1–50. https://doi.org/10.1016/0898-1221(92)90135-5

Lewis, S. N. (2013). *Pragmatic enrichment in language processing and development*. University of Maryland.

Liddy, E. D. (2001). Natural Language Processing. In *Encyclopedia of Library and Information Science* (2nd ed., pp. 1–15). Marcel Decker, Inc.

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

Computation, *14*(11), 2531–2560. https://doi.org/10.1162/089976602760407955

McClelland, J. L., & Rogers, T. T. (2003). The parallel distributed processing approach to semantic cognition. *Nature Reviews Neuroscience*, *4*(4), 310–322. https://doi.org/10.1038/nrn1076

McClelland, J. L., & Rumelhart, D. E. (1985). Distributed Memory and the Representation of General and Specific Information. *Journal of Experimental Psychology: General*, *114*(2), 159–188. https://doi.org/10.1037/0096-3445.114.2.159

Monroe, D. (2014). Neuromorphic computing gets ready for the (really) big time. *Communications of the ACM*, *57*(6), 13–15. https://doi.org/10.1145/2601069

Östling, R. (2018). Part of Speech Tagging: Shallow or Deep Learning? *Northern European Journal of Language Technology*, *5*, 1–15. https://doi.org/10.3384/nejlt.2000-1533.1851

Rabiner, L. R., & Schafer, R. W. (2007). Introduction to digital speech processing. *Foundations and Trends in Signal Processing*, *1*(1–2), 1–194. https://doi.org/10.1561/2000000001

Reiter, E., & Dale, R. (1997). Building applied natural language generation systems. *Natural Language Engineering*, *3*(1), 57–87. https://doi.org/10.1017/S1351324997001502

Rogers, T. T., & McClelland, J. L. (2008). Précis of semantic cognition: A parallel distributed processing approach. *Behavioral and Brain Sciences*, *31*(6), 689–749. https://doi.org/10.1017/S0140525X0800589X

Rooth, M. (1992). A theory of focus interpretation. *Natural Language Semantics*, *1*(1), 75–116. https://doi.org/10.1007/BF02342617

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Schachter, P., & Shopen, T. (2007). Parts-of-speech systems. In T. Shopen (Ed.), *Language Typology and Syntactic Description* (2nd ed., Vol. 1). Cambridge University Press. https://doi.org/10.1017/CBO9780511619427.001

Schaller, R. R. (1997). Moore's Law: past, present, and future. *IEEE Spectrum*, *34*(6), 52–59. https://doi.org/10.1109/6.591665

Sharma, A. (2020). *Top 10 Applications of Natural Language Processing (NLP)*. https://www.analyticsvidhya.com/blog/2020/07/top-10-applications-of-natural-language-processing-nlp/

Smith, E. E., Shoben, E. J., & Rips, L. J. (1974). Structure and process in semantic memory: A featural model for semantic decisions. *Psychological Review*, *81*(3), 214–241.

https://doi.org/10.1037/h0036351

Van der Velde, F. (2020). *Perceptrons*.

Verstraeten, D., Schrauwen, B., D'Haene, M., & Stroobandt, D. (2007). An experimental unification of reservoir computing methods. *Neural Networks*, *20*(3), 391–403. https://doi.org/10.1016/j.neunet.2007.04.003

Waldrop, M. M. (2016). More than moore. *Nature*, *530*(7589). https://link.gale.com/apps/doc/A443132364/HRCA?u=anon~1ee6eafd&sid=HRCA&xid=04326e79

Appendix A: Supplementary results

A1. Learning runs and learning rate

The learning rate is the value that all output connections and the bias are adjusted by. The maximum amount of learning rates determines how often the bias and output connections are adjusted. The program stops trying to converge after this amount of learning runs, or earlier if the desired pattern is learned.

For learning rate, we tested 0.1, 0.01 and 0.001 with a maximum amount of 1000 learning runs. A smaller learning rate was expected to be more precise, because a higher learning rate may cause the outputs to revolve around the right values, but never quite reaching the correct pattern. This 'overshooting' happens, because the output values may differ by only a small amount for each input pattern. When all connection weights are adjusted by the same value, this could mean that multiple output values change from slightly above 0 to slightly below 0. A smaller learning rate could prevent this from happening, but it will take more adjustments to get output values around the threshold. An overview of the overall success rates can be found in table A1. The results show the success vs partial success vs failure rates.

Table A1: Overall success vs partial success vs failure rates for all linear logic gates using single cell perceptron learning, with varying learning rate 0.1, 0.01 and 0.001.

Learning rate	***Success***	***Partial***	***Failure***
0.1	*31 (6,5%)*	*240 (50,0%)*	*209 (43,5%)*
0.01	*21 (4,4%)*	*250 (52,1%)*	*209 (43,5%)*
0.001	*0 (0,0%)*	*271 (56,5%)*	*209 (43,5%)*

The first thing to notice is that the failure rate remains the same for all combinations of learning rate and learning runs. These failures are therefore not attributed to a combination of learning runs and learning rate, but could be explained by other parameters. The second thing to notice is that the success rate of 0.1 is the highest, whereas it was expected that a smaller learning rate would increase the precision and therefore the success rate of the perceptron. For the learning rate of 0.001 – where the success rate is the lowest –, this can be explained by a shortage of learning runs. When the outputs of the cell are too high for any given input, the perceptron algorithm is unable to adjust the connection weights and bias enough to compensate with this amount of learning runs. Therefore, the outputs for [1,1], [1,0] and [0,1]

will all still be positive after all learning runs have completed, meaning the program cannot distinguish between them.

The difference between learning rates 0.1 and 0.01 cannot be explained in a similar way. Both learning rates should suffice for the perceptron to adjust the connection weights enough to get a negative output for any input value. For the 0.1 rate, 10 adjustments could already suffice to reduce the maximum connection strength to a negative, meaning the output activation would be negative as well. For the 0.01 rate, this could take up to 100 runs. After this amount, it is to be expected that the bias might need some more adjusting to reach the correct values, but not up to 1000. A closer inspection of the amount of learning rates it takes for the program to solve for any individual logic gate shows that this is the case (see figure A1). For both learning rates, the perceptron is able to solve mostly within 100 runs. However, with a learning rate of 0.01, there are more iterations that require between 100 and 200 runs to converge. In both cases, there are a few cases in which it takes 300 or more runs.

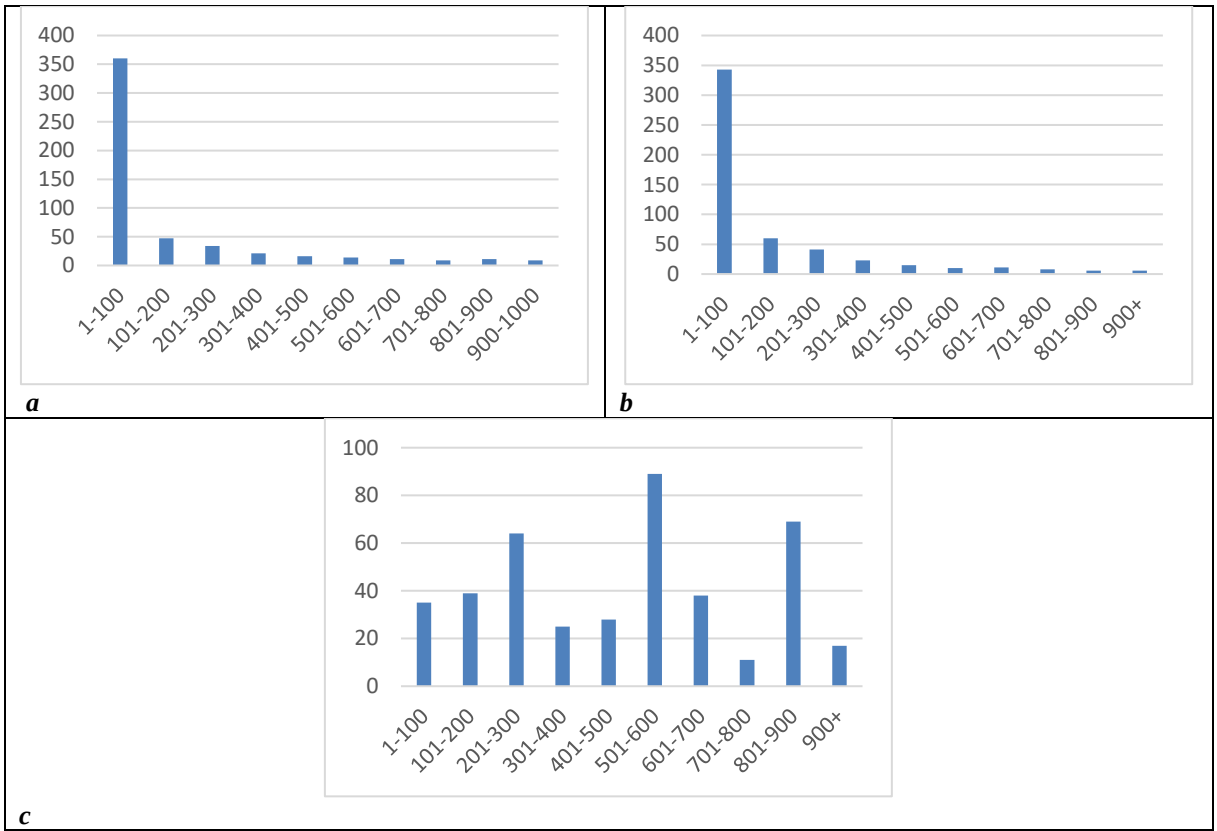


Figure A1: Learning runs at which each logic gate is solved by the single cell perceptron with learning rate 0.1 (a), 0.01 (b) and 0.001 (c). In the cases where the OR logic gate solved, it was always solved at learning run 0 no matter which learning rate was applied. Therefore, this logic gate was left out in this analysis.

While this pattern suggests that a smaller learning rate is actually less precise, the difference between success ratios may be explained by a limitation of the program instead: an

issue with floating points. Floating points (or floats) as a data type are never fully accurate, as they take the closest binary fraction as their value. This means there might be some ‘noise’ on the numbers used for calculation (python.org).

These small additions may make a difference between a positive and a negative output when the actual output value is very close to 0. A closer inspection of the achieved bias and cell outputs shows that this is the case, as there are some values that differ from 0 by a very small amount (e.g., -5.167e-02, -2.000e-05 and -4.297e-02 vs. 1.000e-02 for NOR). This may result in logic gates being solved, while they should not be and vice versa. Since some required internal libraries use floats as their primary data type, this noise is inevitable. Even when rounding down or using the Decimal data type, the inaccuracy of the floating points can influence the outcomes.

As a learning rate of 0.001 is not high enough to reach convergence and the 0.1 learning rate leaves too much room for error, the rest of the analysis was conducted using the 0.01 learning rate condition. While errors could be found in this condition as well, these were smaller in number as in the 0.1 condition, where there might be erroneous convergence even in less than 100 learning runs.

A2. Additional results experiment 1: Perceptron learning with a single reservoir

Cell size

Table A2 shows the success ratio for the cell with different sizes. The number of failures decreases drastically as cell size increases, while partial success shows the opposite pattern. High failure rate in the smallest cells could be explained by a lack of activation flow from input to output nodes.

With a limited number of nodes in the hidden layer, it is likely that the few connections that are formed do not result in pathways from input nodes to the output node, meaning the output node does not get activated, resulting in an output of 0 for all input patterns. The perceptron algorithm only adjusts existing connections rather than generating connections where they do not exist yet, meaning the activation given to the output node does not change. Adding the same bias to all output values is then useless, because all output values remain the same and no distinction can be made.

Table A2: ratio of success vs partial success vs failure for different cell size			
Size	***Success***	***Partial***	***Failure***
10	*1*	*11*	*108*
20	*3*	*50*	*67*
50	*11*	*80*	*29*
100	*6*	*109*	*5*

Adding more nodes to the cell and thereby increasing the number of connections initially seems to increase the success ratio. However, when the cell gets too big, success rate decreases again. In this case, there are so many connections between the cellnodes that activation within the cell is amped up to a maximum value. Spreading this amped up activation level into the output node results in similar activation levels in the output node for all input patterns other than [0,0]. Because of this, the OR pattern is still solvable, as this only requires the perceptron algorithm to distinguish 'any input given' from 'no input given'. However, the AND / NAND gates cannot be solved anymore, as there is no distinction between the other output values.

Sparsity

A similar pattern can be discerned when looking at sparsity (see table A3). Higher sparsity means there are less connections between nodes. As sparsity increases (i.e., the number of connections decreases), so does the failure rate. With fewer connections, the partial success rate decreases. However, the success ratio per sparsity does not resemble the success ratio per cell size. Rather than showing an initial increase and thereafter a decrease, the highest success rate is for the cells with the lowest sparsity threshold.

Table A3: ratio of success vs partial success vs failure for different sparsity			
Sparsity	***Success***	***Partial***	***Failure***
0.80	*6*	*66*	*8*
0.85	*6*	*50*	*24*
0.88	*3*	*48*	*29*
0.92	*6*	*40*	*34*
0.95	*0*	*31*	*49*
0.98	*0*	*15*	*65*

Connection strength

Table A4 shows the success ratio of cells with varying connection strength. The failure rate is fairly consistent for all conditions, but the success and partial successes are not. As strength of connection increases, the success rate declines. Similarly, the partial success rate is higher with stronger connections. It appears that stronger connections make it more difficult for the AND / NAND gates to solve, because the activation in the cell is reaching the upper limit when any input is given, resulting in the same activation level within the cell for input [1,1],

[1,0] and [0,1]. Since the perceptron learning algorithm only adjusts connections from cell- to output nodes, there is no way of adjusting the activation within the cell.

Table A4: ratio of success vs partial success vs failure for different connection strength

Strength of connections	***Success***	***Partial***	***Failure***
0.2	*16*	*53*	*51*
0.5	*2*	*67*	*51*
0.8	*0*	*69*	*51*
Random	*3*	*61*	*56*

When setting the connection strength between nodes randomly, the success rate is halfway between that of 0.5 and 0.8, but the failure rate is slightly higher. When drawing many random numbers between 0 and 1, these numbers should average to approximately 0.5. However, it could be that the lower numbers were attributed to connections between nodes that were not part of the pathway from input to output nodes, whereas some higher numbers were, resulting in a higher connection strength overall.

Seeds

As seen in table A5, using different seeds for generating a network also leads to varying success ratios. It seems that some networks are more suitable for solving the logic gates than others, and that the number of connections does not fully make up for the capabilities of the network.

Table A5: ratio of success vs partial success vs failure for different seeds

Seeds	***Success***	***Partial***	***Failure***
1, 2, 3	*5*	*53*	*38*
4, 5, 6	*6*	*40*	*50*
7, 8, 9	*5*	*49*	*42*
10, 11, 12	*3*	*54*	*39*
13, 14, 15	*2*	*54*	*40*

In some cases (e.g., using seeds 4, 5 and 6), there are not enough pathways for activation flow, resulting in a higher failure rate. Other networks may generate too many pathways between the two input flows, leading to similar activation levels for all input patterns (e.g., seeds 13, 14, 15). This increases the number of partial successes, as such a network is still suitable for solving the OR logic gate.

A3. Additional results experiment 2: Perceptron learning with multiple reservoirs

The second version of the program showed no complete successes. In order to still make some analysis on the effects of the parameters, we compared the results for the linear logic gates

only. This resulted in a success rate as seen in figure A2. The success rates for XOR / XNOR were both too low to make any inferences based on the parameters.

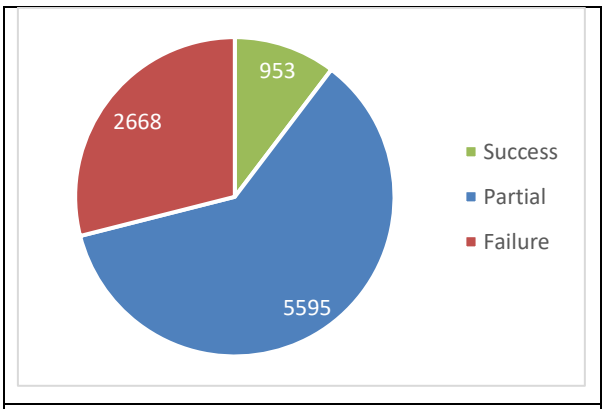


Figure A2: Success ratio of perceptron learning with multiple reservoirs for only linear logic gates.

Cell size

Similar to the single cell iteration, cell size 20 has the highest success rate, followed by cell size 50 (see table A6). These cell settings allow for enough connections to be formed, without inflating activation within the cell. Cell size 10 results in the highest failure rate, suggesting there is no activation flow from input to output. Conversely, cell size 100 results in a high partial success rate. This means the OR / NOR gates likely can be solved, as these only differentiate 'activation' from 'no activation', but the AND / NAND gates cannot be solved due to the similarity in output activation for each given input.

Table A6: ratio of success vs partial success vs failure for different cell size using perceptron learning with multiple reservoirs.

Size	***Success***	***Partial***	***Failure***
10	*226*	*445*	*1633*
20	*378*	*1070*	*856*
50	*246*	*1907*	*151*
100	*103*	*2173*	*28*

Sparsity

The success rates per internal and external sparsity can be found in tables A7 and A8, respectively. For internal sparsity, success rate and partial success rate both decreases as the sparsity value increases (i.e., when there are fewer connections). Conversely, the failure rate increases. This suggests that higher internal sparsity results in too few pathways between input nodes and output node.

External sparsity shows the opposite pattern; as fewer connections are made between subcells, success rate increases. However, failure rate also slightly increases for higher values of external sparsity. It appears that low external sparsity mainly leads to partial successes.

This suggests that low external sparsity settings inflate the activation levels for any given input, making it impossible to distinguish between them. The effect seems more pronounced, but this is likely due to the bigger range of external sparsity settings as compared to internal sparsity.

Table A7: ratio of success vs partial success vs failure for different internal sparsity using perceptron learning with multiple reservoirs.

Int. Spars	***Success***	***Partial***	***Failure***
0.80	*269*	*1219*	*48*
0.85	*174*	*1143*	*219*
0.88	*167*	*1103*	*266*
0.92	*163*	*953*	*420*
0.95	*91*	*650*	*795*
0.98	*89*	*527*	*920*

Table A8: ratio of success vs partial success vs failure for different external sparsity using perceptron learning with multiple reservoirs.

Ext. Spars	***Success***	***Partial***	***Failure***
0.20	*16*	*1116*	*404*
0.50	*33*	*1099*	*404*
0.85	*127*	*989*	*420*
0.90	*174*	*940*	*422*
0.95	*267*	*817*	*452*
1.00	*336*	*634*	*566*

Connection strength

Connection strength also follows the same pattern as in the single cell version; lower connection strength gives a higher success rate (see table A9). It seems that a connection strength exceeding 0.5 is not suitable for this reservoir. However, just like cell size and sparsity, these results cannot be taken at face value, because connection strength impacts the results more when there are more connections within and between the cells.

Table A9: ratio of success vs partial success vs failure for different connection strength using perceptron learning with multiple reservoirs.

Strength of connections	***Success***	***Partial***	***Failure***
0.2	*567*	*1070*	*667*
0.5	*172*	*1471*	*661*
0.8	*53*	*1590*	*661*
Random	*161*	*1464*	*679*

Seeds

The only factor that did not seem to impact the results is the combination of seeds used to generate the network. As table A10 shows, there are slight variations between the success rate of seed combinations, but the maximum difference in successes only accounts for less than 1% of the runs (10,5% success for seeds [1, 2, 3, 4] versus 9,9% for seeds [13, 14, 15, 16]). These numbers are the same for both subcells, as all combinations have been tested.

Table A10: ratio of success vs partial success vs failure for different seeds using perceptron learning with multiple reservoirs.

Seeds	***Success***	***Partial***	***Failure***
[1, 2, 3, 4]	*243*	*1379*	*682*
[5, 6, 7, 8]	*241*	*1407*	*656*
[9, 10, 11, 12]	*240*	*1420*	*644*
[13, 14, 15, 16]	*229*	*1389*	*686*

The results show that the number of pathways generated is of more importance than which pathways. If there are enough pathways from input to output, it appears to matter less that there are some 'dead ends' in the cell layer, especially when one subcell can compensate for the lack of activation in the other subcell.

A4. Additional results experiment 3: Control nodes

Similar to the previous version, the current setup did not result in any full successes. However, the success rates of XOR / XNOR are higher in this version (23 and 16 times, respectively). Therefore, the current section details the results for all linear gates, as well as the results for XOR / XNOR separately. The success rate for only the linear gates can be found in figure A3.

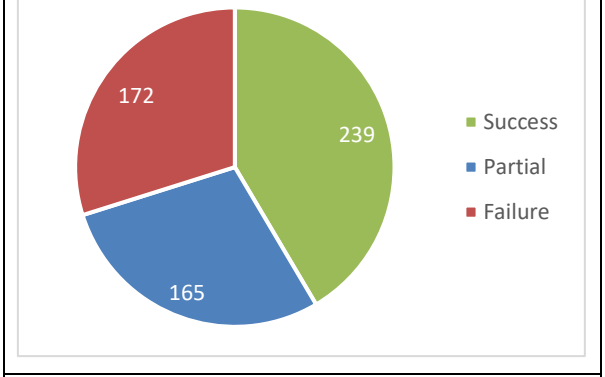


Figure A3: Success ratio of all linear logic gates using control nodes for learning.

The overall success rate for control nodes is much higher than perceptron learning with multiple reservoirs, whereas the partial success rate drastically decreases. This shows that control nodes are better suited for classification of linear logic gates. The percentage of complete failures remains equal, as control nodes cannot make up for a complete lack of activation flow from the input nodes to the output nodes.

Cell size

Table A11 shows the success rate per cell size. Cell sizes of 20 and 50 are still most favorable using control nodes. There are some successes and partial successes for cell size 10, but these rates do not compare to the failure rate. For cell size 100, there are no more failures, but only partial successes. It seems that control nodes are not sufficient to distinguish the activation levels for input [1,1] versus [1,0] and [0,1] when these are too similar.

Table A11: ratio of success vs partial success vs failure of linear logic gates for different cell size using control nodes for learning.

Size	***Success***	***Partial***	***Failure***
10	*17*	*7*	*120*
20	*75*	*21*	*48*
50	*92*	*48*	*4*
100	*55*	*89*	*0*

Table A12: ratio of success of XOR / XNOR for different cell size using control nodes for learning.

Size	***XOR***	***XNOR***
10	*0*	*0*
20	*3*	*2*
50	*13*	*13*
100	*7*	*1*

Table A12 shows the success rate for XOR and XNOR for different cell sizes. Both logic gates are converged most often when the cell size is set to 50. XOR can also solve for cells of size 100, whereas XNOR shows very little successes for any setting other than 50. This suggests that the non-linear logic gates are impacted more than the linear gates by the number of nodes in the system.

Sparsity

Tables A13 and A14 show the results per internal and external sparsity for the linear gates. For the linear gates, it seems that lower internal sparsity leads to higher successes, whereas external sparsity shows the opposite pattern, much like the previous version of the program. It seems that the control nodes are not sufficient to compensate for the activity flow between subcells, as an external sparsity of .85 or higher is required for the number of successes to overtake the number of partial successes.

Table A13: ratio of success vs partial success vs failure for different internal sparsity using control nodes for learning.

Int. Spars	***Success***	***Partial***	***Failure***
0.80	*63*	*33*	*0*
0.85	*49*	*23*	*24*
0.88	*41*	*31*	*24*
0.92	*42*	*30*	*24*
0.95	*32*	*16*	*48*
0.98	*12*	*32*	*52*

Table A14: ratio of success vs partial success vs failure for different external sparsity using control nodes for learning.

Ext. Spars	***Success***	***Partial***	***Failure***
0.20	*11*	*57*	*28*
0.50	*22*	*46*	*28*
0.85	*44*	*24*	*28*
0.90	*46*	*22*	*28*
0.95	*56*	*12*	*28*
1.00	*60*	*4*	*32*

Tables A15 and A16 show the success rates of XOR / XNOR per internal and external sparsity, respectively. The XOR gate seems to favor lower internal sparsity, whereas XNOR solves most often with an internal sparsity of 0.92. External sparsity shows a different pattern for both non-linear gates; XOR is converged more often with higher external sparsity, but XNOR is more dispersed. This suggests that the XOR works better when there are many connections within the subcells, but the activation of subcells have little to no influence on each other. For XNOR, it is harder to determine such a pattern, as the success ratio for both sparsity values are more seemingly random.

Table A15: ratio of success of XOR / XNOR for different internal sparsity using control nodes for learning.

Int. Spars	***XOR***	***XNOR***
0.80	*16*	*0*
0.85	*2*	*1*
0.88	*4*	*4*
0.92	*1*	*9*
0.95	*0*	*2*
0.98	*0*	*0*

Table A16: ratio of success of XOR / XNOR for different external sparsity using control nodes for learning.

Ext. Spars	***XOR***	***XNOR***
0.20	*1*	*0*
0.50	*0*	*1*
0.85	*2*	*4*
0.90	*2*	*4*
0.95	*4*	*5*
1.00	*14*	*2*

Connection strength

Tables A17 and A18 show the success ratios per connection strength for the linear gates and XOR / XNOR, respectively. For the linear gates, full success rate decreases as connection strength increases. Activation is likely reaching a maximum value no matter the input with higher connection strength. Random connection strength leads to the lowest success rate, although it should be expected to more closely resemble a static strength of 0.5, given that the random numbers should average to that value.

The failure rate is the same for all connection strengths. This is unsurprising, as the OR gate can solve so long as there is any activation reaching the output node, no matter how strong the activation is. Connection strength has no effect on the number of pathways between input and output nodes, so varying this value should not impact the failure rate at all.

Table A17: ratio of success vs partial success vs failure of linear logic gates for connection strength using control nodes for learning.

Strength of connections	***Success***	***Partial***	***Failure***
0.2	*83*	*18*	*43*
0.5	*63*	*38*	*43*
0.8	*51*	*50*	*43*
Random	*42*	*59*	*43*

Table A18: ratio of success of XOR / XNOR for different connection strength using control nodes for learning.

Strength of connections	***XOR***	***XNOR***
0.2	*13*	*4*
0.5	*4*	*6*
0.8	*2*	*6*
Random	*4*	*0*

The patterns for XOR / XNOR look a bit different. XOR also performs best with low connection strength, but the success rate for 0.5 and random connections are the same. This matches the expectations, but not the pattern for the linear gates. For XNOR, the pattern is less clear; there are successful configurations for each connection strength, except for random connections. Interestingly, the success rates for 0.5 and 0.8 are equal and higher than the success rate of the lowest connection strength, contrasting the success rates of all other logic gates.

A5. Additional results experiment 4: Control nodes with split input

Like the previous versions, the success rate of XOR / XNOR are low (2 successes for XOR, 24 for XNOR). Therefore, the analysis will contain the linear gates and XOR / XNOR separately. The success rate of the linear gates can be found in figure A4.

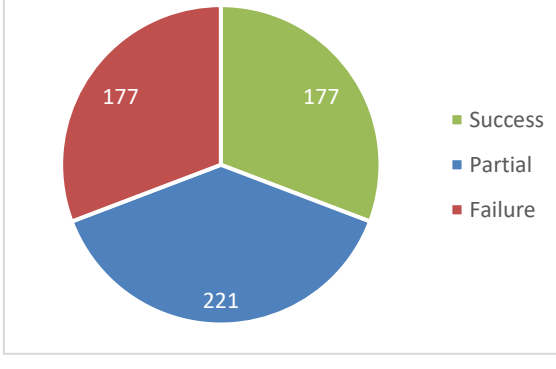


Figure A4: Success ratio of all linear logic gates using control nodes and split input streams.

The overall success rate of linear gates has dropped in comparison to the previous version, as did the success rate of XOR. XNOR, on the other hand, performed slightly better in this version. The failure rate for linear gates is approximately the same as in the previous version, but the partial success rate is much higher.

Cell size

The success rates per cell size can be found in table A19 (for linear gates) and table A20 (for XOR / XNOR). Whereas there was a significant difference in the success rates for the linear gates between cell size 20 and 50 in the previous version, the current version performs equally for cell sizes 20, 50 and 100. Interestingly, cell size 20 now shows a slightly higher success rate than cell size 50. Cell size 10 still shows more complete failures than partial or full successes and underperforms greatly as compared to the other configurations.

Table A19: ratio of success vs partial vs failure of linear gates for different cell size using control nodes for learning and split input streams.

Size	***Success***	***Partial***	***Failure***
10	*21*	*3*	*120*
20	*59*	*33*	*52*
50	*52*	*87*	*5*
100	*45*	*98*	*0*

Table A20: ratio of success of XOR / XNOR for different cell size using control nodes for learning and split input streams.

Size	***XOR***	***XNOR***
10	*0*	*0*
20	*0*	*15*
50	*0*	*5*
100	*2*	*4*

For XOR, there is no pattern to be found, as there are only two successful configurations. It is interesting to note that these configurations were cell size 100, contrasting previous versions where cell size 50 was the better setting, but the number is too small to

make any meaningful interpretations. For XNOR, cell size 20 seems most favorable, whereas cell size 50 was most successful in the previous version. This development makes sense, considering the overall reservoir is twice as big due to the splitting into multiple subcells.

Sparsity

Tables A21 and A22 show the results per internal and external sparsity for the linear gates. Similar to the previous version, lower internal sparsity and high external sparsity leads to higher successes. Even when the input streams are split, it still seems favorable for the input streams to have as little connections between them as possible, and to have many connections within one input stream to ensure activation flow from input to output nodes.

Table A21: ratio of success vs partial success vs failure for different internal sparsity using split input streams.

Int. Spars	***Success***	***Partial***	***Failure***
0.80	*56*	*40*	*0*
0.85	*28*	*44*	*24*
0.88	*31*	*41*	*24*
0.92	*30*	*38*	*28*
0.95	*23*	*24*	*49*
0.98	*9*	*34*	*52*

Table A22: ratio of success vs partial success vs failure for different external sparsity using split input streams.

Ext. Spars	***Success***	***Partial***	***Failure***
0.20	*4*	*63*	*28*
0.50	*8*	*60*	*28*
0.85	*26*	*42*	*28*
0.90	*32*	*36*	*28*
0.95	*48*	*20*	*28*
1.00	*59*	*0*	*37*

For XNOR, the pattern looks quite similar; low internal sparsity and high external sparsity leads to more successes (see tables A23 and A24). However, having no connections between input streams appears to break this pattern. It should be noted that there is little variance between the success rates of 0.85, 0.90 and 0.95, and the success rate is relatively low. Therefore, this pattern may just exist out of chance, and if the success rate were higher, the pattern could look entirely different. For XOR, nothing notable can be stated, as the success rate is too low.

Table A23: ratio of success of XOR / XNOR for different internal sparsity using split input streams.

Int. Spars	***XOR***	***XNOR***
0.80	*0*	*11*
0.85	*2*	*7*
0.88	*0*	*5*
0.92	*0*	*0*
0.95	*0*	*0*
0.98	*0*	*1*

Table A24: ratio of success of XOR / XNOR for different external sparsity using split input streams.

Ext. Spars	***XOR***	***XNOR***
0.20	*0*	*1*
0.50	*0*	*2*
0.85	*0*	*6*
0.90	*1*	*7*
0.95	*1*	*8*
1.00	*0*	*0*

Connection strength

The results per connection strength can be found in tables A25 and A26 for the linear gates and XOR / XNOR, respectively. For the linear gates, the pattern looks similar to the previous version; success rate decreases as connection strength increases. The failure rate is constant,

except for the ‘random’ condition, where there is one extra failure. This is most likely due to the floating-point issue discussed in paragraph A1, as connection strength should not impact the failure rate at all.

Table A25: ratio of success vs partial success vs failure of linear logic gates for connection strength using split input streams.

Strength of connections	***Success***	***Partial***	***Failure***
0.2	*58*	*42*	*44*
0.5	*48*	*52*	*44*
0.8	*37*	*63*	*44*
Random	*34*	*64*	*45*

Table A26: ratio of success of XOR / XNOR for different connection strength using split input streams.

Strength of connections	***XOR***	***XNOR***
0.2	*2*	*14*
0.5	*0*	*5*
0.8	*0*	*3*
Random	*0*	*2*

For XOR, both successes were present in the lowest connection strength setting, much like the previous version. The pattern for XNOR looks different as compared to the previous version. In the current version, the success rate decreases as connection strength increases, whereas the success rate was more spread out in the previous version. It is possible that this version shows a more pronounced effect due to the increase in successes overall.

A6. Additional results experiment 5: Saturation cells

The saturation cells made it possible for certain configurations to reach full successes on all linear and non-linear gates. However, the success rates for XOR / XNOR were still relatively low compared to the linear gates (35 successful XOR, 45 for XNOR). Therefore, the following analysis will follow the same pattern as those of the previous versions in comparing the linear gates and the XOR / XNOR separately. The success rate of the linear gates can be found in figure A5.

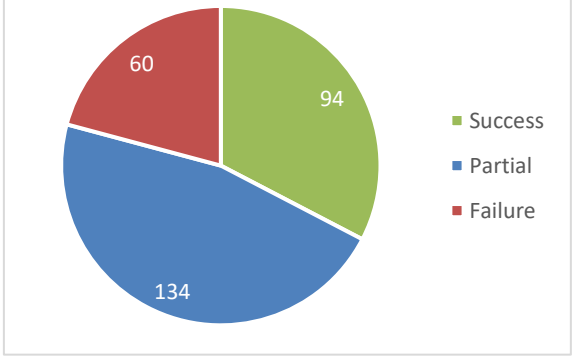


Figure A5: Success ratio of all linear logic gates using saturation cells.

As mentioned in section 9, the current program was only tested using cell sizes 20 and 50, in part because these were the most successful and in part due to time constraints. The results indicate a bigger overall success rate and a smaller number of failures, but this is partly

because only the most successful cell sizes were considered. Therefore, a comparison of overall results between this version and the previous program cannot effectively be made. However, it is still possible to check what the effects of all parameters are on the success rates, and to see if these patterns match that of previous versions.

Cell size

Tables A27 and A28 show the success rates per cell size for the linear gates and for XOR / XNOR. For the linear logic gates, there is not a lot of difference in complete successes between cell size 20 and 50, but the partial success and failure rates differ greatly. For cell size 20, the partial success and failure rates are close together and the failure rate slightly exceeds the partial success rate. With a cell size of 50, there are only few configurations for which the cell cannot converge on at least one logic gate.

Table A27: ratio of success vs partial success vs failure of linear logic gates for different cell size using saturation cells.

Size	***Success***	***Partial***	***Failure***
20	*48*	*43*	*53*
50	*46*	*91*	*7*

Table A28: ratio of success of XOR / XNOR for different cell size using saturation cells.

Size	***XOR***	***XNOR***
20	*12*	*17*
50	*23*	*28*

For XOR / XNOR, cell size 50 seems favorable; the success rates for cell size 50 is almost twice as big than those of cell size 20 for both logic gates. This pattern meets expectations, as bigger cells allow for more pathways to form between input and output. When the cell size is set to 20, it is possible for one or multiple reservoirs to lack activation flow, which makes it impossible for XOR / XNOR to solve.

Sparsity

The pattern for internal and external sparsity looks similar to previous versions for the linear gates (see table A29 for internal sparsity and A30 for external sparsity). Success rate decreases with higher internal sparsity and higher internal sparsity results in failures, as there is no activation flow possible. For external sparsity, success rates increase as external sparsity does. Low external sparsity does not lead to more failures, but to more partial successes, as the OR gate is easier to solve when there is a lot of activation flow.

Table A29: ratio of success vs partial success vs failure for different internal sparsity using saturation cells.

Int. Spars	***Success***	***Partial***	***Failure***
0.80	*30*	*18*	*0*
0.85	*20*	*28*	*0*
0.88	*15*	*33*	*0*
0.92	*17*	*26*	*5*
0.95	*11*	*12*	*25*
0.98	*1*	*17*	*30*

Table A30: ratio of success vs partial success vs failure for different external sparsity using saturation cells.

Ext. Spars	***Success***	***Partial***	***Failure***
0.20	*0*	*40*	*8*
0.50	*1*	*39*	*8*
0.85	*19*	*21*	*8*
0.90	*22*	*17*	*9*
0.95	*24*	*14*	*10*
1.00	*28*	*3*	*17*

The pattern for XOR / XNOR looks similar as well; they both perform better with lower internal sparsity and higher external sparsity (see tables A31 and A32). This pattern does not match that of previous versions, as the success rates were more dispersed. It appears that the saturation cells bring some consistency to the cell, making certain settings more fitting for all logic gates.

Table A31: ratio of success of XOR / XNOR for different internal sparsity using saturation cells.

Int. Spars	***XOR***	***XNOR***
0.80	*19*	*23*
0.85	*10*	*13*
0.88	*3*	*4*
0.92	*3*	*5*
0.95	*0*	*0*
0.98	*0*	*0*

Table A32: ratio of success of XOR / XNOR for different external sparsity using saturation cells.

Ext. Spars	***XOR***	***XNOR***
0.20	*0*	*0*
0.50	*0*	*0*
0.85	*5*	*5*
0.90	*6*	*8*
0.95	*7*	*11*
1.00	*17*	*21*

Connection strength

The success rates per connection strength can be found in tables A33 (linear gates) and A34 (XOR / XNOR). For the linear logic gates, the connection strength does not seem to impact the results as much. Connection strength 0.5 has a slightly higher success rate as compared to 0.2 and 0.8, and random connection strength has the lowest success rate overall. However, the difference between the conditions in terms of absolute numbers is low.

Table A33: ratio of success vs partial success vs failure of linear logic gates for connection strength using split input streams.

Strength of connections	***Success***	***Partial***	***Failure***
0.2	*23*	*32*	*17*
0.5	*26*	*32*	*14*
0.8	*24*	*34*	*14*
Random	*21*	*36*	*15*

Table A34: ratio of success of XOR / XNOR for different connection strength using split input streams.

Strength of connections	***XOR***	***XNOR***
0.2	*8*	*14*
0.5	*11*	*12*
0.8	*7*	*8*
Random	*9*	*11*

The same can be said for the success rates of XOR; it is difficult to pinpoint a pattern based on the low variability of the results. Similar to the results of the linear gates, connection strength 0.5 seems most favorable. For XNOR, the success rate seems to decrease as

connection strength increases, similar to the previous version of the program. However, the difference between success rates for conditions is still relatively small. Random connection strength results in a similar success rate as the 0.5 condition, suggesting the average random connection strength is close to this value.

A7. Additional results experiment 6: Semantic network

The semantic network program was run in two conditions, namely the single seed and the unique seed settings. The single seed setting refers to all reservoirs (representing attributes) having the same seed combination, namely [1,2,3,0] for input to reservoir, within reservoir, reservoir to output, and between reservoirs, respectively. The unique seed setting refers to each reservoir having a unique seed combination, starting with [1,2,3,0] for attribute 1, [4,5,6,0] for attribute 2 etc. External sparsity was not used in this experiment, which is also the reason the last seed number was not varied.

The results in the following sections do not only focus on success vs. partial success vs. failure rates. As a partial success is already achieved when as few as one attribute solved with a configuration, it is not informative to display these results. Rather, we also show how many attributes are solved by each cell setting, as a partial success configuration solving 13 attributes should be valued higher than a configuration that solved only one. In total, both seed conditions ran a total of 96 configurations solving 14 attributes.

Cell size

For the single seed configurations, the success vs partial success vs failure rate per cell size can be found in table A35. Table A36 shows the total number of attributes solved by each cell size setting, as well as the average number of attributes solved by each setting. Each cell size setting solved a total of 336 attributes (24 configurations * 14 attributes).

Success rate initially increases with cell size, but the biggest cells perform slightly worse than cell size 50 with one more failure instead of a success. Cell size 10 and 20 both show more failures than successes. Reservoirs with only 10 nodes are hardly capable of solving the AND gate. Only twice is there a full success for all attributes, and another two configurations are capable of solving only the attributes that are true for all items (see section 10 – Experiment 6).

When looking at the number of attributes solved by each cell size setting, it becomes more evident that cells of size 10 are too small for running a semantic network. With an average of 1,4 attributes solved by each configuration, it performs much worse than any other condition. However, this could be due to the specific seed combination that was used to

generate all attributes. It is possible that other seed combinations allow for better pathways to form, even reservoirs with only 10 nodes. The same can be said for the difference between cell size 50 and 100. Although cell size 50 solves more attributes on average, the actual difference is only one iteration (14 attributes). A different seed setting may favor cell size 100 over cell size 50, or further increase the performance of cell size 50 instead.

Table A35: ratio of success vs partial success vs failure all attributes for different cell size using the same seeds for all attributes. Success means all attributes were solved using the same parameter settings, partial means some but not all attributes solved, failure means none of the attributes reached convergence.

Size	***Success***	***Partial***	***Failure***
10	*2*	*2*	*20*
20	*9*	*4*	*11*
50	*15*	*1*	*8*
100	*14*	*1*	*9*

*Table A36: Number of solved attributes per cell size using the same seed for all attributes. Percentages are based on the total number of attributes attempted per cell size setting (i.e., 14 attributes * 24 configurations = 336 attributes per cell size). Average per configuration is calculated by dividing the total number solved by configurations in each condition.*

Size	***Attributes solved***		***Average per configuration***
10	*34*	*(10,1%)*	*1,4*
20	*138*	*(41,1%)*	*5,8*
50	*213*	*(63,4%)*	*8,9*
100	*199*	*(59,2%)*	*8,3*

Table A37 shows the success rate per attribute split by cell size. The attributes that are true for all items (i.e., *'animal', 'move'* and *'skin'*) are solved more often than other attributes in all conditions. This suggests that the relation input has a bigger impact on the total output activation (see also section 10). However, the effect is less pronounced for cells of size 50 and 100, suggesting that ensuring there are enough connections is sufficient to counter the effect of the relation input having a larger impact on the total output activation.

Table A37: number of successful convergences per attribute per cell size using the same seed for every attribute.

Attribute	***Cell size 10***	***Cell size 20***	***Cell size 50***	***Cell size 100***
animal	*4*	*13*	*16*	*15*
bird	*2*	*9*	*15*	*14*
fish	*2*	*9*	*15*	*14*
move	*4*	*13*	*16*	*15*
skin	*4*	*13*	*16*	*15*
feathers	*2*	*9*	*15*	*14*
fly	*2*	*9*	*15*	*14*
wings	*2*	*9*	*15*	*14*
red	*2*	*9*	*15*	*14*
sing	*2*	*9*	*15*	*14*
yellow	*2*	*9*	*15*	*14*
swim	*2*	*9*	*15*	*14*
gills	*2*	*9*	*15*	*14*
scales	*2*	*9*	*15*	*14*

In the unique seed configurations, there were no complete successes. Table A38 shows the success ratios for unique seeds per cell size setting, table A39 shows the total number of solved attributes per cell size. The partial success rate increases with larger cell sizes, meaning there are less failures. For cell size 100, there are no configurations of parameters for which every attribute fails to converge. However, there is considerable difference in the (average) number of attributes each cell size configuration is capable of solving. Cell size 10 only solves an average of 1,8 attributes per configuration, whereas cell size 50 solves almost 9 attributes on average. While cell size 100 has the highest rate of partial successes, this setting solves less attributes on average than the cell size 50 setting.

Table A38: ratio of success vs partial success vs failure all attributes for different cell size using unique seeds for all reservoirs.

Size	***Success***	***Partial***	***Failure***
10	*0*	*15*	*9*
20	*0*	*20*	*4*
50	*0*	*21*	*3*
100	*0*	*24*	*0*

Table A39: Number of solved attributes per cell size using unique seeds per attribute. Percentages and averages are calculated in the same manner as in table A36.

Size	***Attributes solved***		***Average per configuration***
10	*44*	*(13,1%)*	*1,8*
20	*158*	*(47,0%)*	*6,6*
50	*209*	*(62,2%)*	*8,7*
100	*176*	*(52,4%)*	*7,3*

Table A40 shows the success rate per attribute split by cell size for the unique seed condition. While there is no discernable pattern for most attributes, these results show that cell size 10 only performs well on 2 attributes (*'skin'* and *'wings'*) that, together, account for about half of its total successes. Conversely, there are six attributes that were never solved by reservoirs with a size of 10 nodes.

Table A40: number of successful convergences per attribute per cell size using a unique seed for each attribute.

Attribute	***Cell size 10***	***Cell size 20***	***Cell size 50***	***Cell size 100***
animal	*4*	*15*	*19*	*18*
bird	*0*	*4*	*13*	*13*
fish	*0*	*5*	*11*	*14*
move	*4*	*12*	*10*	*18*
skin	*8*	*17*	*18*	*13*
feathers	*0*	*15*	*18*	*10*
fly	*0*	*10*	*17*	*10*
wings	*15*	*19*	*20*	*13*
red	*4*	*16*	*9*	*12*
sing	*4*	*8*	*17*	*12*
yellow	*0*	*9*	*15*	*5*
swim	*1*	*12*	*20*	*19*
gills	*0*	*9*	*14*	*10*
scales	*4*	*7*	*8*	*9*

Internal sparsity

Table A41 shows the success vs partial success vs failure rate per sparsity setting for the single seed configurations. Table A42 shows the total and average number of attributes solved by each sparsity setting in the same seed condition. Every setting of sparsity solved a total of 224 attributes (16 configurations * 14 attributes).

For the single seed settings, success rate decreases as sparsity increases. The highest sparsity setting results in only failures, suggesting there are not enough pathways from input to output nodes to solve the AND gate. The only outlier in this pattern is sparsity 0.95, which performs slightly better than sparsity 0.92. This is also seen in the number of attributes solved by each sparsity setting. Higher sparsity means less attributes are solved, except for the 0.95 sparsity settings.

Table A41: ratio of success vs partial success vs failure all attributes for different sparsity using the same seeds for all attributes.

Sparsity	***Success***	***Partial***	***Failure***
0.80	*10*	*2*	*4*
0.85	*10*	*1*	*5*
0.88	*8*	*0*	*8*
0.92	*5*	*4*	*7*
0.95	*7*	*1*	*8*
0.98	*0*	*0*	*16*

Table A42: Number of solved attributes per cell size using the same seeds for every attribute.

Sparsity	***Attributes solved***		***Average per configuration***
0.80	*146*	*(65,2%)*	*9,1*
0.85	*143*	*(63,8%)*	*8,9*
0.88	*112*	*(50,0%)*	*7*
0.92	*82*	*(36,6%)*	*5,1*
0.95	*101*	*(45,1%)*	*6,3*
0.98	*0*	*(0,0%)*	*0*

For the unique seeds condition, the results per sparsity setting can be found in tables A43 (success ratio) and A44 (number of attributes solved). Similar to the single seed settings, failure rate increases for the unique seed settings as the sparsity threshold is higher. Every setting up to .92 results in partial successes for every configuration, but as sparsity increases, the failure rate starts to exceed the partial success rate. However, there are some parameter configurations using sparsity 0.98 for which some attributes are solved, unlike in the single seed setting. This pattern is mimicked by the number of attributes solved by each sparsity setting. The average number of attributes solved drops from almost 9 with the lowest sparsity setting to barely not 1,5 in the highest setting.

Table A43: ratio of success vs partial success vs failure all attributes for different sparsity using unique seeds for all reservoirs.

Sparsity	***Success***	***Partial***	***Failure***
0.80	*0*	*16*	*0*
0.85	*0*	*16*	*0*
0.88	*0*	*16*	*0*
0.92	*0*	*15*	*1*
0.95	*0*	*12*	*4*
0.98	*0*	*5*	*11*

Table A44: Number of solved attributes per cell size using unique seeds per attribute.

Sparsity	***Attributes solved***		***Average per configuration***
0.80	*137*	*(61,1%)*	*8,6*
0.85	*134*	*(59,8%)*	*8,4*
0.88	*112*	*(50,0%)*	*7*
0.92	*91*	*(40,6%)*	*5,8*
0.95	*90*	*(40,1%)*	*5,6*
0.98	*23*	*(10,3%)*	*1,4*

The difference between 0.80 and 0.95 is not as big as between 0.95 and 0.98 (21,0% between 0.80 and 0.95 versus 29,8% between 0.95 and 0.98). It seems that the biggest cutoff point for success is between the two highest sparsity settings. Table A45 shows the number of times each attribute was solved per sparsity setting. These results show that there are 7 attributes for which sparsity 0.98 did not solve under any configuration. Interestingly, 6 of these attributes match the attributes that were not solvable using cell size 10 (see table A39). There are no other discernable patterns with regards to which attributes are solved by which setting of sparsity. The distribution of successes in each sparsity setting is not equal for every attribute, and the individual attributes often deviate from the overall pattern in which a lower sparsity means a higher success rate.

Table A45: number of successful convergences per attribute per sparsity using a unique seed for each attribute.

Attribute	***Spars. 0.80***	***Spars. 0.85***	***Spars. 0.88***	***Spars. 0.92***	***Spars. 0.95***	***Spars. 0.98***
animal	*15*	*10*	*12*	*11*	*8*	*0*
bird	*8*	*7*	*6*	*6*	*3*	*0*
fish	*7*	*8*	*3*	*8*	*4*	*0*
move	*11*	*8*	*8*	*6*	*7*	*4*
skin	*16*	*12*	*8*	*10*	*9*	*1*
feathers	*8*	*11*	*9*	*8*	*7*	*0*
fly	*5*	*10*	*10*	*4*	*8*	*0*
wings	*12*	*16*	*13*	*12*	*10*	*4*
red	*10*	*5*	*6*	*8*	*8*	*4*
sing	*13*	*10*	*5*	*4*	*5*	*4*
yellow	*4*	*10*	*4*	*6*	*5*	*0*
swim	*13*	*10*	*12*	*4*	*8*	*5*
gills	*8*	*7*	*10*	*1*	*7*	*0*
scales	*7*	*10*	*6*	*3*	*1*	*1*

Connection strength

For the single seed configurations, the success ratio per setting of connection strength can be found in table A46. The total and average number of attributes solved per setting are shown in table A47. Each setting of connection strength solved 24 configurations * 14 attributes, resulting in a total of 336 attributes being solved per group.

Contrary to the results of experiment 3, success rate seems to increase as connection strength is higher. This is unexpected, as higher connection strength usually resulted in amplified output activation, meaning there was not enough difference in the activation for input [1,1] versus [1,0] and [0,1]. One possible explanation is the lack of external connections between reservoirs in this version of the program. Therefore, the activation within another reservoir has no impact on the activation in each individual reservoir. This could result in more variety in output activation, allowing for the AND gate to solve.

The current trend, in which higher connection strength leads to more successes, can also be seen in the number of attributes solved per setting of connection strength. However, even the best result (for connection strength 0.8) barely leads to solving more than half of the attributes on average per configuration. Random connection strength results in a success rate that falls between 0.5 and 0.8. This is in line with expectations, as the randomly generated numbers should average to 0.5, and stronger connections have a bigger impact on the total activation of the cell.

Table A46: ratio of success vs partial success vs failure for different connection strength using the same seeds for all attributes.

Strength of connections	***Success***	***Partial***	***Failure***
0.2	*7*	*2*	*15*
0.5	*10*	*2*	*12*
0.8	*12*	*2*	*10*
Random	*11*	*2*	*11*

Table A47: Number of solved attributes per connection strength setting using the same seeds for all attributes.

Strength of connections	***Attributes solved***	***Average per configuration***
0.2	*104 (31,0%)*	*4,3*
0.5	*146 (43,5%)*	*6,1*
0.8	*174 (51,8%)*	*7,3*
Random	*160 (47,6%)*	*6,7*

Tables A48 and A49 show the success ratio and the number of attributes solved per connection strength setting for configurations using a unique seed for each attribute. Contrary to the single seed settings, the unique seed setting show no differences in the success ratio between the different connection strength conditions. Every setting of connection strength results in 20 partial successes and 4 failures. There are some differences in the number of attributes solved per connection strength setting, but these are very minimal. The 'random' connection strength condition appears to perform slightly worse as compared to the others, but with an average of 0,6 less attributes solved per configuration, this difference seems negligible.

Table A48: ratio of success vs partial success vs failure for different connection strength using unique seeds for all attributes.

Strength of connections	***Success***	***Partial***	***Failure***
0.2	*0*	*20*	*4*
0.5	*0*	*20*	*4*
0.8	*0*	*20*	*4*
Random	*0*	*20*	*4*

Table A49: Number of solved attributes per connection strength setting using unique seeds for all attributes.

Strength of connections	***Attributes solved***	***Average per configuration***
0.2	*152 (45,2%)*	*6,3*
0.5	*147 (43,8%)*	*6,1*
0.8	*152 (45,2%)*	*6,3*
Random	*136 (40,5%)*	*5,7*

The same pattern (or lack thereof) can be found when splitting the results per attribute (see table A50). The distribution of the success rates of each attribute are approximately equal for every setting of connection strength. That is, each individual attribute seems to perform equally well no matter what setting of connection strength is used. There are some exceptions, such as *'bird'* in the random connection strength setting, which performs much worse as

compared to the other settings. Another notable exception is *'red'*, which is the only attribute that appears to perform better with random connection strength as compared to other configurations.

Table A50: number of successful convergences per attribute per connection strength using a unique seed for each attribute.

Attribute	***Connection strength 0.2***	***Connection strength 0.5***	***Connection strength 0.8***	***Connection strength random***
animal	*14*	*13*	*15*	*14*
bird	*9*	*9*	*9*	*3*
fish	*8*	*8*	*8*	*6*
move	*11*	*12*	*11*	*10*
skin	*14*	*13*	*15*	*14*
feathers	*10*	*10*	*11*	*12*
fly	*11*	*9*	*10*	*7*
wings	*17*	*16*	*17*	*17*
red	*10*	*10*	*9*	*12*
sing	*11*	*10*	*10*	*10*
yellow	*8*	*8*	*8*	*5*
swim	*13*	*13*	*13*	*13*
gills	*9*	*8*	*9*	*7*
scales	*7*	*8*	*7*	*6*

A8. Conclusion

Overall, cell sizes (or cluster sizes) of 20 and 50 seem most favorable for all logic gates to solve. A lower number of nodes in the reservoir will result in few pathways from input nodes through the network to the output node, with a relatively high chance of no complete pathways existing at all. This means that one or both input streams do not reach the output node, making it impossible to learn any logic gate. This results in a high failure rate.

Cell size 100, on the other hand, shows the opposite pattern; there may be too many pathways in the cell that influence each other, causing an inflation of activation. At some point, all connected nodes might have the same activation level (in part due to the squashing function), meaning a distinction between the activation for input patterns [1,1] versus [1,0] and [0,1] is not possible anymore. The OR / NOR gates can still be solved when these activation levels are the same, but AND / NAND cannot. Therefore, this results in a high partial success rate.

Interestingly, for the semantic network, cell size 100 is the second-best setting for both the single seed for all attributes and unique seed per attribute settings. Cell size 50 is capable of solving the most attributes on average per configuration of parameters, but reservoirs with 100 nodes perform almost as well. This is counterintuitive, as the semantic network only solves AND gates and the initial results show that cells of size 100 usually result in partial

successes. However, given the variety in internal pathways due to 14 different seed combinations, it is not unthinkable that some of these reservoirs are more capable of solving the AND gate than others, resulting in a high success rate per reservoir.

The same pattern exists for (internal) sparsity; as sparsity increases, the success rate decreases. This goes for solving logic gates as well as the semantic network. With a high sparsity setting, there are few connections from each node to the others. This makes it difficult for complete pathways to form, meaning the input may not reach the output node. Conversely, when the sparsity setting is low, there could be too many connections between nodes, leading to an inflated activation level. The current results may not reflect this fact well enough, as we only tested an internal sparsity of 0.80 and up, resulting in the highest success rate for the lowest settings of internal sparsity. However, these settings also lead to the highest partial success ratio, which is in line with the given theory.

For external sparsity, success rate universally increases as there are fewer connections between reservoirs (experiments 2 and 3), or between input streams (experiments 4 and 5). When the activation of one cell or one input stream heavily influences the activation of another, the partial success increases. This suggests a similar pattern to lower internal sparsity; activation becomes inflated and, by extension, too similar to differentiate. Most cells are only capable of solving linear gates when external sparsity is set to .85 and up, with the best setting consistently being 1 (i.e., no connections between reservoirs / input streams).

There seems to be a similar effect for connection strength as well. As connection strength increases, the success rate drops for every version of the program, except for the saturation cells and the semantic network. With low connection strength, it might be possible to dampen the inflation of activation within the cell, and thereby the overall activation. Random connection strength generally leads to the lowest success rate, possible because the stronger connections have a more pronounced effect on the reservoir than the weaker connections. Therefore, activation might still be inflated.

In the semantic network, it appears that equal contribution from both input nodes still makes it possible to solve the AND gate even with higher connection strength. Rather, the effect is in the opposite direction. This might have to do with the absence of activation flow between reservoirs. The activation within a reservoir is therefore not amped up by the activation of another reservoir, resulting in a stronger impact of the input activation on the total output as connection strength increases.

The seed parameter showed the lowest difference in success rates between each setting when solving the logic gates. In order to successfully solve all logic gates, the number of connections (as a function of cell size and sparsity) and the strength of these connections seems more important than which specific pathways are formed. However, as seen in the semantic network (section A7), this does not mean that the seeds do not influence the capabilities of the network at all. It is possible for a cell with very few connections to solve, given the right connections are formed. Similarly, it is possible for a cell with many connections to reach a full success, if the formed pathways do not influence each other too much. This could be the effect of the correct seed settings.

Given these results, it may appear simple to pinpoint the ‘best settings’ of the cell by taking a cell size of 20 or 50, a relatively low internal sparsity, high external sparsity and low connection strength, there is no guarantee that such a cell would be fit to solve all logic gates by definition. It is possible that the interactions between the parameters show a different pattern overall, that is hard to determine due to the many factors contributing the overall fitness. However, these results do give an indication of which configurations have a lower chance of success, namely those with too little or too many potential connections within the reservoir.

Appendix B: Source codes

The first program consists out of a single class (Nodenetwork). All other versions consist of two classes; CellNetwork and Subcell.

B1. Perceptron learning with a single reservoir

```
1. # -*- coding: utf-8 -*-
2. """
3. Created on Tue Apr 28 12:08:19 2020
4. Adaptation: 01-08-2020
5.
6. Versionnumber: 3.92
7.
8. @authors: Kevin Liu & Robbin Koopman
9. with help of Frank van der Velde
10.  based on the boron dopant cell by Chen et al. (2020):
   https://doi.org/10.1038/s41586-019-1901-0
11.  """
12.
13.  import numpy as np
14.  import random
15.  import copy
16.  import xlwt
17.  import math
18.
19.  class Nodenetwork():
20.
21.      def __init__(self, _cellsize, _sparsity, _seedcombo,
   _connectionstrength):
22.          # amount of input- and outputnodes.
23.          # for logic gates that do not require a hidden layer; 2
   inputnodes and 1 outputnode.
24.          inputnodes = 2
25.          outputnodes = 1
26.
27.          # skipvalue that determines when not to spread activation
   from node x to node y
28.          self.skip = -99
29.          # amount of timesteps used for spreading of activation
30.          self.timesteps = 10
31.
32.          # array for storing all activation levels of all cellnodes
   at each timestep
33.          # for each inputpattern
34.          self.cellnodeactivations = np.zeros(4 * self.timesteps *
   _cellsize).reshape(
35.                                              4, self.timesteps,
   _cellsize)
36.
37.          # arrays for storing connectionstrengths from node x to
   node y
38.          self.inputconnections = self.networkgen(inputnodes,
   _cellsize,
39.                                                  _seedcombo[0],
   _sparsity, _connectionstrength)
40.          self.cellconnections = self.networkgen(_cellsize,
   _cellsize,
```

```
41.                                                 _seedcombo[1],
   _sparsity, _connectionstrength, True)
42.          self.outputconnections = self.networkgen(_cellsize,
   outputnodes,
43.                                                   _seedcombo[2],
   _sparsity, _connectionstrength)
44.
45.      """
46.      Generates a network consisting of connections between
   _startnodes and _endnodes.
47.      Each _startnode can be connected to any number of _endnodes.
48.      Connectionstrength is the same for all connections; this can be
   changed later if needed.
49.      self.skip indicates no connection is made between _startnode[i]
   and _endnode[j].
50.
51.      _cellmidlayer is only True when connecting cellnodes, as
   cellnodes should not be connected to themselves.
52.      In all other cases, it is false; e.g. inputnode 0 could be
   connected to cellnode 0.
53.
54.      Returns the generated network.
55.      """
56.      def networkgen(self, _startnodes, _endnodes, _seed, _threshold,
   _connstrength = 0, _cellmidlayer = False):
57.          dummyconnections = np.zeros(_startnodes *
   _endnodes).reshape(_startnodes, _endnodes)
58.          random.seed(_seed)
59.          for i in range(_startnodes):
60.              for j in range(_endnodes):
61.                  dummyconnections[i,j] = self.skip
62.                  if _cellmidlayer and j == i:
63.                          continue
64.                  if random.random() >= _threshold:
65.                      if not _connstrength == 0.0:
66.                          dummyconnections[i, j] = _connstrength
67.                      else:
68.                          dummyconnections[i, j] = random.random()
69.          return dummyconnections
70.
71.      """
72.      Calculates the total activation onto cellnode q, given by:
73.      - the inputnode * connectionstrength of inputnote k to cellnode
   q
74.      - activation of cellnodes k at timestep t connected to cellnode
   q * connectionstrength of cellnode k to cellnode q
75.      Stores the squashed inputSum as activation of cellnodes q in
   timestep t+1 in
76.      cellnodeactivations for this input (indicated by _inputnumber)
77.      """
78.      def spreadactivation(self, _input, _inputnumber):
79.          for t in range(self.timesteps-1):
80.              nextstep = t + 1
81.              for q in
   range(len(self.cellnodeactivations[_inputnumber, t])):
82.                  inputSum = 0.0
83.                  for k in range(len(_input)):
84.                      if self.inputconnections[k, q] != self.skip:
85.                          inputSum += _input[k] *
   self.inputconnections[k, q]
```

```
86.                     for k in
   range(len(self.cellnodeactivations[_inputnumber, t])):
87.                         if self.cellconnections[k, q] != self.skip:
88.                             inputSum +=
   self.cellnodeactivations[_inputnumber, t, k] *
   self.cellconnections[k, q]
89.                     self.cellnodeactivations[_inputnumber, nextstep, q]
   = math.tanh(inputSum)
90.
91.         """
92.         Calculates the output of the cell;
93.         Takes the activations of each cellnode at the last timestep
   (stored in self.cellnodeactivations per input),
94.         multiplies each activation by the connectionstrength of
   cellnode q on outputnode,
95.         and adds all the (activations*connectionstrengths) together.
96.
97.         Returns the sum of all (activations*connectionstrengths)
98.         """
99.         def getoutput(self, _inputnumber, _outputconnections):
100.            outputSum = 0.0
101.            for k in range(len(self.cellnodeactivations[_inputnumber, -
   1])):
102.                if _outputconnections[k, 0] != self.skip:
103.                    outputSum += self.cellnodeactivations[_inputnumber,
   -1, k] * _outputconnections[k, 0]
104.            return outputSum
105.
106.        """
107.        Makes a deep copy of outputconnections (so the original is
   never edited by the program).
108.        For each learningrun until convergence is reached:
109.        - Calculates summedoutputs by adding outputs of both cells
   together for outputpattern; 1 for >0, 0 for <=0.
110.        - Compares desiredpattern with outputpattern and adjusts bias
   and outputconnections when there is a mismatch.
111.
112.        Returns the summedoutputs of the cell, the learningrun at which
   convergence was reached and the achieved bias.
113.        """
114.        def perceptronlearning(self, _desiredpattern, _learningruns =
   1000, _learningrate = 0.01):
115.            inputpatterns = [ [1,1],
116.                              [1,0],
117.                              [0,1],
118.                              [0,0] ]
119.            for i in range(len(inputpatterns)):
120.                self.spreadactivation(inputpatterns[i], i)
121.
122.            standinoutputconnections =
   copy.deepcopy(self.outputconnections)
123.            print(standinoutputconnections)
124.            biasnode = -1
125.            bias = 0.0
126.            outputpattern = np.zeros(len(_desiredpattern))
127.
128.            for q in range(_learningruns):
129.                # print("current learningrun: " + str(q))
130.                summedoutputs = np.zeros(len(_desiredpattern))
131.                for i in range(len(_desiredpattern)): # For each
   potential input pattern
```

```
132.                 summedoutputs[i] = self.getoutput(i,
    standinoutputconnections) + (bias * biasnode)
133.                 # print("Celloutputs@ input#" + str(i) + ": " +
    str(summedoutputs[i]))
134.                 if summedoutputs[i] > 0:
135.                     outputpattern[i] = 1
136.                 else:
137.                     outputpattern[i] = 0
138.
139.             error = 0.0
140.             convergence = q
141.             for p in range(len(_desiredpattern)):
142.                 error = _desiredpattern[p] - outputpattern[p]
143.                 if error != 0:
144.                     convergence = None
145.                     bias += error * biasnode * _learningrate
146.                     # print("new bias: " + str(bias))
147.                     for w in range(len(standinoutputconnections)):
148.                         if standinoutputconnections[w, 0] !=
    self.skip:
149.                             standinoutputconnections[w, 0] += error
    * self.cellnodeactivations[p, -1, w] * _learningrate
150.                             # print(" ")
151.             if convergence != None:
152.                 print("inp")
153.                 print (self.inputconnections)
154.                 print("cell")
155.                 print(self.cellconnections)
156.                 print("out")
157.                 print(standinoutputconnections)
158.                 print("summedoutputs")
159.                 print(summedoutputs)
160.                 break
161.         return [convergence, bias, summedoutputs]
162.
163.     """
164.     Writes the results of the learn function into an excel file.
165.     Sheet 1 shows the success vs partial vs failure rates of the
    cell with varying size, sparsity, conn_strength and seed
166.     Sheet 2 contains detailed information per logic gate;
    learningruns for reaching convergence, bias and output activations
167.     """
168.     def addresultstoexcel(self, _excelinfo, _networkinfo, _results,
    _row):
169.         # _excelinfo = [workbook, sheet1, sheet2]
170.         # _networkinfo = [cellsize, sparsity, conn_str, seeds]
171.         # _results = [logicgate, convergencerun, bias, outputs] per
    logicgate
172.         for i in range(len(_networkinfo)):
173.             _excelinfo[1].write(_row, i, str(_networkinfo[i]))
174.
175.         allsuccess = True
176.         allfailed = True
177.         for pattern in range(len(_results)):
178.             if _results[pattern][1] == None:
179.                 allsuccess = False
180.             else:
181.                 allfailed = False
182.         if allsuccess:
183.             _excelinfo[1].write(_row, len(_networkinfo), "success")
184.         elif allfailed:
```

```
185.             _excelinfo[1].write(_row, len(_networkinfo), "failure")
186.         else:
187.             _excelinfo[1].write(_row, len(_networkinfo), "partial")
188.
189.         for j in range(len(_results)):
190.             subrow = ((_row-1) * len(_results)) + j + 1
191.             for i in range(len(_networkinfo)):
192.                 _excelinfo[2].write(subrow, i,
    str(_networkinfo[i]))
193.             for k in range(len(_results[j])):
194.                 _excelinfo[2].write(subrow, len(_networkinfo) + k,
    str(_results[j][k]))
195.
196. """
197. Preparing the Excel output file with two sheets:
198.     First slide containing information on overall success rate per
    configuration
199.     Second slide containing information on convergence per logic
    gate per configuration
200. """
201.
202. wb = xlwt.Workbook()
203. sh1 = wb.add_sheet("successrate")
204. sh1.write(0,0,"cellsize")
205. sh1.write(0,1,"sparsity")
206. sh1.write(0,2,"conn_strength")
207. sh1.write(0,3,"seeds")
208. sh1.write(0,4,"result")
209.
210. sh2 = wb.add_sheet("logicgates")
211. sh2.write(0,0,"cellsize")
212. sh2.write(0,1,"sparsity")
213. sh2.write(0,2,"conn_strength")
214. sh2.write(0,3,"seeds")
215. sh2.write(0,4,"logicgate")
216. sh2.write(0,5,"converged_at_run")
217. sh2.write(0,6,"bias")
218. sh2.write(0,7,"celloutputs")
219.
220. resultscounter = 1
221.
222. """
223. Loops through the program for each cellsize, sparsity,
    inputsparsity and pattern.
224. Initializes a nodenetwork and spreads activation.
225. Stores the results of the learning function into a list and
    forwards this to the write function.
226. """
227. cellsize = [10, 20, 50, 100]
228. seedcombos = [ [1, 2, 3],
229.                [4, 5, 6],
230.                [7, 8, 9],
231.                [10,11,12],
232.                [13,14,15] ]
233. sparsity = [.80, .85, .88, .92, .95, .98]
234. conn_strength = [0.2, 0.5, 0.8, 0.0] # 0.0 for random
235.
236.
237. # Desired output pattern of perceptron; lists have to contain the
    same amount of elements
238. patternnames = ["OR", "NOR", "AND", "NAND", "XOR", "XNOR"]
```

```
239.
240. """
241. Returns the desired output for the given pattern.
242. """
243. def get_pattern(_patternname):
244.     if _patternname == "AND":
245.         return [1,0,0,0]
246.     if _patternname == "NAND":
247.         return [0,1,1,1]
248.     if _patternname == "OR":
249.         return [1,1,1,0]
250.     if _patternname == "NOR":
251.         return [0,0,0,1]
252.     if _patternname == "XOR":
253.         return [0,1,1,0]
254.     if _patternname == "XNOR":
255.         return [1,0,0,1]
256.
257. for size in range(len(cellsize)):
258.     for spars in range(len(sparsity)):
259.         for seeds in range(len(seedcombos)):
260.             for strength in range(len(conn_strength)):
261.
262.                 print("Current iteration: " + str(resultscounter))
263.                 networkinfo = [cellsize[size], sparsity[spars],
    conn_strength[strength], seedcombos[seeds]]
264.                 results = []
265.                 for p in range(len(patternnames)):
266.                     nodenetwork = Nodenetwork(cellsize[size],
    sparsity[spars], seedcombos[seeds], conn_strength[strength])
267.                     results.append([*[patternnames[p]],
    *nodenetwork.perceptronlearning(get_pattern(patternnames[p]))])
268.                 nodenetwork.addresultstoexcel([wb, sh1, sh2],
    networkinfo, results, resultscounter)
269.                 wb.save("results_v1.xls")
270.                 resultscounter += 1
```

B2. Perceptron learning with multiple reservoirs

CellNetwork

```
1. # -*- coding: utf-8 -*-
2. """
3. Created on Wed Jun 10 13:17:22 2020
4.
5. @author: Kevin Liu & Robbin Koopman
6. """
7.
8. from Subcell import Subcell
9. import numpy as np
10.  import xlwt
11.
12.  class CellNetwork():
13.      """
14.      Initiates two subcells required for intermediate logic gate
    solving.
15.      Stores the subcells into self.subcells so they can be called on
    by other functions.
16.      """
```

```
17.      def __init__(self, _cellsize, _seedcombos, _intsparsity,
   _extsparsity, _connstr):
18.          self.timesteps = 10
19.          self.subcells = np.ndarray(2, dtype = Subcell)
20.          for c in range(len(self.subcells)):
21.              dopantcell = Subcell(_cellsize, _seedcombos[c],
   self.timesteps, _intsparsity, _extsparsity, _connstr)
22.              self.subcells[c] = dopantcell
23.
24.      """
25.      Spreads activation into both subcells.
26.      For a pre-determined amount of learningruns:
27.          - Calls learn_subcell for intermediate logic gate solving
   (until the respective gate is solved).
28.          - Adjusts the bias of the overall cell until either the
   logic gate converged or the max amount
29.            of learning runs has been reached.
30.      Returns the results of the learning algorithm:
31.          - intermediate logic gate, convergence run, outputs per
   subcell
32.          - convergence run, bias, outputs of overall cell
33.      """
34.      def main(self, _patternname):
35.          learningruns = 1000
36.          learningrate = 0.01
37.          biasnode = -1   # biasnode value (constant)
38.          bias = 0.0
39.          desiredpatterns = self.get_patterns(_patternname)
40.          inputpatterns = [ [1,1],
41.                            [1,0],
42.                            [0,1],
43.                            [0,0] ]
44.
45.          results = []
46.          sub1_results = []
47.          sub2_results = []
48.
49.          for i in range(len(inputpatterns)):
50.              for t in range(self.timesteps-1):
51.                  self.subcells[0].spreadactivation(t,
   inputpatterns[i], i, self.subcells[1])
52.                  self.subcells[1].spreadactivation(t,
   inputpatterns[i], i, self.subcells[0])
53.
54.          for run in range(learningruns):
55.              learned = run
56.              sub1_results =
   self.subcells[0].learn_subpattern(desiredpatterns[1], learningrate)
57.              sub2_results =
   self.subcells[1].learn_subpattern(desiredpatterns[2], learningrate)
58.
59.              celloutputs = np.zeros(len(inputpatterns))
60.              outputpattern = np.zeros(len(inputpatterns))
61.
62.              for i in range(len(celloutputs)):
63.                  celloutputs[i] = sub1_results[2][i] +
   sub2_results[2][i] + (bias * biasnode)
64.                  if celloutputs[i] > 0:
65.                      outputpattern[i] = 1
66.
67.              for i in range(len(desiredpatterns[0])):
```

```
68.                     error = desiredpatterns[0][i] - outputpattern[i]
69.                     if error != 0:
70.                         bias += round(error * biasnode * learningrate,
   4)
71.                         learned = None
72.
73.                 if learned != None:
74.                     print(_patternname + " learned at run: " +
   str(run))
75.                     break
76.                 elif run == learningruns-1:
77.                     print(_patternname + " failed to converge.")
78.
79.             results.append(desiredpatterns[1])
80.             for r in range(len(sub1_results)):
81.                 results.append(sub1_results[r])
82.             results.append(desiredpatterns[2])
83.             for r2 in range(len(sub2_results)):
84.                 results.append(sub2_results[r2])
85.             results.append(learned)
86.             results.append(bias)
87.             results.append(celloutputs)
88.
89.             return results
90.
91.         """
92.         Requires the name of the overall pattern to be solved.
93.         Returns the overall pattern and the intermediate patterns.
94.         (e.g. returns [ [0,1,1,0], [1,1,1,0], [0,1,1,1] ] for XNOR)
95.         """
96.         def get_patterns(self, _patternname):
97.             pos = 1
98.             neg = 0
99.             if _patternname == "AND":
100.                networkpattern  = [pos,neg,neg,neg]
101.                sub1pattern     = [pos,neg,neg,neg]
102.                sub2pattern     = [pos,neg,neg,neg]
103.            elif _patternname == "OR":
104.                networkpattern  = [pos,pos,pos,neg]
105.                sub1pattern     = [pos,pos,pos,neg]
106.                sub2pattern     = [pos,pos,pos,neg]
107.            elif _patternname == "NAND":
108.                networkpattern  = [neg,pos,pos,pos]
109.                sub1pattern     = [neg,pos,pos,pos]
110.                sub2pattern     = [neg,pos,pos,pos]
111.            elif _patternname == "NOR":
112.                networkpattern  = [neg,neg,neg,pos]
113.                sub1pattern     = [neg,neg,neg,pos]
114.                sub2pattern     = [neg,neg,neg,pos]
115.            elif _patternname == "XOR":
116.                networkpattern  = [neg,pos,pos,neg]
117.                sub1pattern     = [pos,pos,pos,neg]
118.                sub2pattern     = [neg,pos,pos,pos]
119.            elif _patternname == "XNOR":
120.                networkpattern  = [pos,neg,neg,pos]
121.                sub1pattern     = [pos,neg,neg,neg]
122.                sub2pattern     = [neg,neg,neg,pos]
123.            return [networkpattern, sub1pattern, sub2pattern]
124.
125. """
126. Writes the results of the current iteration into an Excel file.
```

```
First sheet is overall success:
    Writes networkinfo in the correct columns.
    Checks convergence runs of all logic gates;
    all None = failed, no None = success, otherwise partial.
Second sheet is success per logic gate:
    Calculate the row to write on based on _row and the amount of logic gates solved.
    Write all data in the corresponding column.
"""
def addresultstoexcel(_excelinfo, _networkinfo, _results, _row):
    # _networkinfo = [id, cellsize, sparsity, conn_str, seeds]
    # _results =    [logicgate, desiredpatterns[1], subcell1convergence, subcell1outputs,
    #               desiredpatterns[2], subcell2convergence, subcell2outputs,
    #               learned, bias_cell, celloutputs] per logicgate
    for i in range(len(_networkinfo)):
        _excelinfo[1].write(_row, i, str(_networkinfo[i]))

    allsuccess = True
    allfailed = True
    for pattern in range(len(_results)):
        if _results[pattern][9] == None:
            allsuccess = False
        else:
            allfailed = False
    if allsuccess:
        _excelinfo[1].write(_row, len(_networkinfo), "success")
    elif allfailed:
        _excelinfo[1].write(_row, len(_networkinfo), "failure")
    else:
        _excelinfo[1].write(_row, len(_networkinfo), "partial")

    for j in range(len(_results)):
        subrow = ((_row-1) * len(_results)) + j + 1
        for i in range(len(_networkinfo)):
            _excelinfo[2].write(subrow, i, str(_networkinfo[i]))
        for k in range(len(_results[j])):
            _excelinfo[2].write(subrow, len(_networkinfo) + k, str(_results[j][k]))

"""
Preparing the Excel file.
Sheet 1 is overall success (per iteration, all logic gates).
Sheet 2 is success per logic gate.
"""
wb = xlwt.Workbook()
sh1 = wb.add_sheet("successrate")
sh1.write(0,0,"id")
sh1.write(0,1,"cellsize")
sh1.write(0,2,"intsparsity")
sh1.write(0,3,"extsparsity")
sh1.write(0,4,"conn_strength")
sh1.write(0,5, "seed1")
sh1.write(0,6, "seed2")
sh1.write(0,7,"result")

sh2 = wb.add_sheet("logicgates")
sh2.write(0,0,"id")
sh2.write(0,1,"cellsize")
sh2.write(0,2,"intsparsity")
```

```
184. sh2.write(0,3,"extsparsity")
185. sh2.write(0,4,"conn_strength")
186. sh2.write(0,5, "seed1")
187. sh2.write(0,6, "seed2")
188. sh2.write(0,7,"logicgate_cell")
189. sh2.write(0,8,"subgate1")
190. sh2.write(0,9,"sub1convergence")
191. sh2.write(0,10, "sub1bias")
192. sh2.write(0,11,"sub1outputs")
193. sh2.write(0,12,"subgate2")
194. sh2.write(0,13,"sub2convergence")
195. sh2.write(0,14, "sub2bias")
196. sh2.write(0,15,"sub2outputs")
197. sh2.write(0,16,"cellconvergence")
198. sh2.write(0,17,"cellbias")
199. sh2.write(0,18,"summedoutputs")
200.
201. """
202. Loops through the program for each cellsize, sparsity,
   inputsparsity and pattern.
203. Initializes a nodenetwork and spreads activation.
204. Stores the results of the learning function into a list and
   forwards this to the write function.
205. """
206. desiredpatterns = ["AND","NAND","OR","NOR", "XOR", "XNOR"]
207. cellsize        = [10, 20, 50, 100]
208. intsparsity     = [.80, .85, .88, .92, .95, .98]
209. extsparsity     = [1, .95, .90, .85, .50, .20]
210. conn_strength   = [0.2, 0.5, 0.8, 0.0] # 0.0 for random
211. seedcombos      = [ [1,2,3,4],
212.                     [5,6,7,8],
213.                     [9,10,11,12],
214.                     [13,14,15,16] ]
215. resultscounter = 1
216.
217. for size in range(len(cellsize)):
218.     for intspars in range(len(intsparsity)):
219.         for extspars in range(len(extsparsity)):
220.             for strength in range(len(conn_strength)):
221.                 for seeds in range(len(seedcombos)):
222.                     for seeds2 in range(len(seedcombos)):
223.                         print("Current iteration: " +
   str(resultscounter))
224.                         print("Cellnetwork with nodes=" +
   str(cellsize[size]) + ", intspars=" +
225.                                 str(intsparsity[intspars]) + ",
   extspars=" + str(extsparsity[extspars]) +
226.                                 ", conn_str=" +
   str(conn_strength[strength]))
227.
228.                         results = []
229.                         networkinfo = [resultscounter,
   cellsize[size],
230.                                        intsparsity[intspars],
   extsparsity[extspars],
231.                                        conn_strength[strength],
   seedcombos[seeds], seedcombos[seeds2]]
232.                         for p in range(len(desiredpatterns)):
233.                             cellnetwork =
   CellNetwork(cellsize[size], [seedcombos[seeds], seedcombos[seeds2]],
```

```
234.
   intsparsity[intspars], extsparsity[extspars],
235.
   conn_strength[strength])
236.                                results.append([*[desiredpatterns[p]],
237.
   *cellnetwork.main(desiredpatterns[p])])
238.
239.                           addresultstoexcel([wb, sh1, sh2],
   networkinfo, results, resultscounter)
240.                           wb.save("results_v4.87.xls")
241.                           print("results saved")
242.                           resultscounter += 1
```

Subcell

```
1. # -*- coding: utf-8 -*-
2. """
3. Created on Tue Apr 28 12:08:19 2020
4. Adaptation: 30-06-2020
5.
6. @author: Kevin Liu & Robbin Koopman
7. """
8.
9. import numpy as np
10.  import random
11.  import math
12.
13.  class Subcell():
14.
15.      def __init__(self, _cellnodes, _seedcombo, _timesteps,
   _intsparsity, _extsparsity, _connstr, _inputnodes = 2):
16.          # _seedcombo = [input, cell, output, external]
17.          outputnodes = 1
18.          self.skip = -99
19.
20.          # array for storing all activation levels of all cellnodes
   at each timestep
21.          # for each inputpattern
22.          self.activation_cellnodes = np.zeros(4 * _timesteps *
   _cellnodes).reshape(
23.                                               4, _timesteps,
   _cellnodes)
24.
25.          # Generating connection arrays
26.          self.inputconnections = self.networkgen(_inputnodes,
   _cellnodes, _seedcombo[0], _intsparsity, _connstr)
27.          self.nodeconnections = self.networkgen(_cellnodes,
   _cellnodes, _seedcombo[1], _intsparsity, _connstr, _cellmidlayer =
   True)
28.          self.outputconnections = self.networkgen(_cellnodes,
   outputnodes, _seedcombo[2], _intsparsity, _connstr)
29.          self.intercellconnections = self.networkgen(_cellnodes,
   _cellnodes, _seedcombo[3], _extsparsity, _connstr)
30.
31.          self.bias = 0.0
32.
33.      """
34.      Generates a network consisting of connections between
   _startnodes and _endnodes.
```

```
35.       Each _startnode can be connected to any number of _endnodes.
36.       Connectionstrength is the same for all connections; this can be
   changed later if needed.
37.       self.skip indicates no connection is made between _startnode[i]
   and _endnode[j].
38.
39.       _cellmidlayer is only True when connecting cellnodes
   internally, as cellnodes should not be connected to themselves.
40.       In all other cases, it is false; e.g. inputnode 0 could be
   connected to cellnode 0; cellnode 0 could be connected to cellnode 0
   of another cell.
41.
42.       Returns the generated network.
43.       """
44.       def networkgen(self, _startnodes, _endnodes, _seed, _threshold,
   _connstrength = 0, _cellmidlayer = False):
45.           dummyconnections = np.zeros(_startnodes *
   _endnodes).reshape(_startnodes, _endnodes)
46.           random.seed(_seed)
47.           for i in range(_startnodes):
48.               for j in range(_endnodes):
49.                   dummyconnections[i,j] = self.skip
50.                   if _cellmidlayer and j == i:
51.                           continue
52.                   if random.random() >= _threshold:
53.                       if not _connstrength == 0.0:
54.                           dummyconnections[i, j] = _connstrength
55.                       else:
56.                           dummyconnections[i, j] = random.random()
57.           return dummyconnections
58.
59.       """
60.       Calculates the total activation onto cellnode q, given by:
61.       - the inputnode * connectionstrength of inputnote k to cellnode
   q
62.       - activation of cellnodes k at timestep t connected to cellnode
   q * connectionstrength of cellnode k to cellnode q
63.       Stores the squashed inputSum as activation of cellnodes q in
   timestep t+1 in
64.       cellnodeactivations for this input (indicated by _inputnumber)
65.       """
66.       def spreadactivation(self, _timestep, _input, _inputnumber,
   _othercell=None):
67.           if _othercell != None:
68.               inputothercell =
   _othercell.activation_cellnodes[_inputnumber, _timestep]
69.           nextstep = _timestep + 1
70.           for q in range(len(self.activation_cellnodes[_inputnumber,
   _timestep])):
71.               inputsum = 0.0
72.               for k in range(len(_input)):
73.                   if self.inputconnections[k, q] != self.skip:
74.                       inputsum += _input[k] *
   self.inputconnections[k, q]
75.               for k in
   range(len(self.activation_cellnodes[_inputnumber, _timestep])):
76.                   if self.nodeconnections[k, q] != self.skip:
77.                       inputsum +=
   self.activation_cellnodes[_inputnumber, _timestep, k] *
   self.nodeconnections[k, q]
78.               if _othercell != None:
```

```
                for k in range(len(inputothercell)):
                    if _othercell.intercellconnections[k, q] !=
self.skip:
                        inputsum += inputothercell[k] *
_othercell.intercellconnections[k, q]

            self.activation_cellnodes[_inputnumber, nextstep, q] =
round(math.tanh(inputsum),4)

    """
    Calculates the output of the cell;
    Takes the activations of each cellnode at the last timestep
(stored in self.cellnodeactivations per input),
    multiplies each activation by the connectionstrength of
cellnode q on outputnode,
    and adds all the (activations*connectionstrengths) together.

    Returns the sum of all (activations*connectionstrengths)
    """
    def getoutput(self, _inputnumber):
        outputsum = 0.0
        for k in range(len(self.activation_cellnodes[_inputnumber,
-1])):
            if self.outputconnections[k, 0] != self.skip:
                outputsum +=
self.activation_cellnodes[_inputnumber, -1, k] *
self.outputconnections[k, 0]
        return round(outputsum, 4)

    """
    Solves the intermediate logic gates.
    Requires the pattern to be solved, the cell to solve the
pattern and the learningrate.
    Adjusts the output connections of the subcell and the bias.
    Returns True when the subcell converged, the newly adjusted
bias, and the outputs of the subcell.
    """
    def learn_subpattern(self, _desiredpattern, _learningrate):
        learned = True
        biasnode = -1
        celloutputs = np.zeros(len(_desiredpattern))
        outputpattern = np.zeros(len(_desiredpattern))
        for i in range(len(celloutputs)):
            celloutputs[i] = round(self.getoutput(i) + (self.bias *
biasnode),4)
            if celloutputs[i] > 0:
                outputpattern[i] = 1

        for i in range(len(_desiredpattern)):
            error = _desiredpattern[i] - outputpattern[i]
            if error != 0:
                learned = False
                self.bias += round(error * biasnode *
_learningrate, 5)
                self.adjustweights(i, round((error *
_learningrate),5))
        return [learned, self.bias, celloutputs]

    def adjustweights(self, _inputnumber, _adjustment):
        for w in range(len(self.outputconnections)):
            if self.outputconnections[w, 0] != self.skip:
```

```
127.                 self.outputconnections[w, 0] +=
   self.activation_cellnodes[_inputnumber, -1, w] * _adjustment
128.                 if self.outputconnections[w, 0] <= self.skip:    #
   use for now: ensure that skip is not connection weight
129.                     self.outputconnections[w, 0] = self.skip + 1
```

B3. Learning with control nodes

CellNetwork

```
1. # -*- coding: utf-8 -*-
2. """
3. Created on Wed Jun 10 13:17:22 2020
4.
5. @author: Kevin Liu & Robbin Koopman
6. """
7.
8. from Subcell import Subcell
9. import numpy as np
10.  import xlwt
11.
12.  class CellNetwork():
13.
14.      def __init__(self, _cellsize, _intspars, _extspars,
   _seedcombos, _connstr):
15.          self.timesteps = 10
16.          self.controlactivationcell = 0.0
17.
18.          self.subcells = np.ndarray(2, dtype = Subcell)
19.          for c in range(len(self.subcells)):
20.              dopantcell = Subcell(_cellsize, _seedcombos[c],
   _intspars, _extspars, _connstr)
21.              self.subcells[c] = dopantcell
22.
23.      def main(self, _patternname):
24.          learningruns = 1000
25.          learningrate = 0.01
26.          despatterns = self.get_patterns(_patternname)
27.          inputpatterns = np.array(  [     [ 1, 1 ],
28.                                           [ 1, 0 ],
29.                                           [ 0, 1 ],
30.                                           [ 0, 0 ] ] )
31.
32.          results = []
33.          sub1_results = []
34.          sub2_results = []
35.
36.          for q in range(learningruns):
37.              errvalue = 0.0
38.              learned = True
39.              summedoutputs = np.zeros(len(despatterns[0]))
40.              outputpattern = np.zeros(len(despatterns[0]))
41.
42.              for i in range(len(inputpatterns)):
43.                  for t in range(self.timesteps-1):
44.                      self.subcells[0].spreadactivation(t,
   inputpatterns[i], i, self.subcells[1])
45.                      self.subcells[1].spreadactivation(t,
   inputpatterns[i], i, self.subcells[0])
46.
```

```
47.             sub1_results =
    self.subcells[0].learn_subpattern(inputpatterns, despatterns[1],
    learningrate)
48.             sub2_results =
    self.subcells[1].learn_subpattern(inputpatterns, despatterns[2],
    learningrate)
49.
50.             for i in range(len(despatterns[0])):
51.                 summedoutputs[i] = (sub1_results[2][i] +
    sub2_results[2][i] +
52.                                     self.controlactivationcell)
53.                 if summedoutputs[i] > 0:
54.                     outputpattern[i] = 1
55.
56.             for i in range(len(despatterns[0])):
57.                 error = despatterns[0][i] - outputpattern[i]
58.                 if error != 0:
59.                     errvalue = error
60.                     learned = False
61.
62.             if errvalue != 0:
63.                 self.controlactivationcell =
    round(self.controlactivationcell + learningrate * errvalue, 4)
64.
65.             if learned:
66.                 print("Network learned after " + str(q) + "
    learningruns.")
67.                 learned = q
68.                 break
69.             elif q == learningruns-1:
70.                 print("Network failed to converge.")
71.                 learned = None
72.
73.         results.append(_patternname)
74.         results.append(despatterns[1])
75.         for r in range(len(sub1_results)):
76.             results.append(sub1_results[r])
77.         results.append(despatterns[2])
78.         for r2 in range(len(sub2_results)):
79.             results.append(sub2_results[r2])
80.
81.         results.append(learned)
82.         results.append(self.controlactivationcell)
83.         results.append(summedoutputs)
84.
85.         return results
86.
87.     """
88.     Requires the name of the overall pattern to be solved.
89.     Returns the overall pattern and the intermediate patterns.
90.     (e.g. returns [ [0,1,1,0], [1,1,1,0], [0,1,1,1] ] for XNOR)
91.     """
92.     def get_patterns(self, _patternname):
93.         if _patternname == "AND":
94.             desiredpattern = [1,0,0,0]
95.             subpattern1 = [1,0,0,0]
96.             subpattern2 = [1,0,0,0]
97.         elif _patternname == "OR":
98.             desiredpattern = [1,1,1,0]
99.             subpattern1 = [1,1,1,0]
100.            subpattern2 = [1,1,1,0]
```

```
        elif _patternname == "NOR":
            desiredpattern = [0,0,0,1]
            subpattern1 = [0,0,0,1]
            subpattern2 = [0,0,0,1]
        elif _patternname == "NAND":
            desiredpattern = [0,1,1,1]
            subpattern1 = [0,1,1,1]
            subpattern2 = [0,1,1,1]
        elif _patternname == "XOR":
            desiredpattern = [0,1,1,0]
            subpattern1 = [1,1,1,0]
            subpattern2 = [0,1,1,1]
        elif _patternname == "XNOR":
            desiredpattern = [1,0,0,1]
            subpattern1 = [1,0,0,0]
            subpattern2 = [0,0,0,1]

        return [desiredpattern, subpattern1, subpattern2]

"""
Preparing the Excel file.
Sheet 1 is overall success (per iteration, all logic gates).
Sheet 2 is success per logic gate.
"""
wb = xlwt.Workbook()
sh1 = wb.add_sheet("successrate")
sh1.write(0,0,"id")
sh1.write(0,1,"cellsize")
sh1.write(0,2,"intsparsity")
sh1.write(0,3,"extsparsity")
sh1.write(0,4,"conn_strength")
sh1.write(0,5,"result")

sh2 = wb.add_sheet("logicgates")
sh2.write(0,0,"id")
sh2.write(0,1,"cellsize")
sh2.write(0,2,"intsparsity")
sh2.write(0,3,"extsparsity")
sh2.write(0,4,"conn_strength")
sh2.write(0,5,"logicgate_cell")
sh2.write(0,6,"subgate1")
sh2.write(0,7,"sub1convergence")
sh2.write(0,8,"c3_activation")
sh2.write(0,9,"sub1outputs")
sh2.write(0,10,"subgate2")
sh2.write(0,11,"sub2convergence")
sh2.write(0,12,"c4_activation")
sh2.write(0,13,"sub2outputs")
sh2.write(0,14,"cellconvergence")
sh2.write(0,15,"c5_activation")
sh2.write(0,16,"summedoutputs")
print("Excel file created")

"""
Writes the results of the current iteration into an Excel file.
First sheet is overall success:
    Writes networkinfo in the correct columns.
    Checks convergence runs of all logic gates;
    all None = failed, no None = success, otherwise partial.
Second sheet is success per logic gate:
```

```
    Calculate the row to write on based on _row and the amount of logic gates solved.
    Write all data in the corresponding column.
"""
def addresultstoexcel(_excelinfo, _networkinfo, _results, _row):
    # _excelinfo    = [workbook, sheet1, sheet2]
    # _networkinfo  = [cellsize, sparsity, conn_str, seeds]
    # _results      = [logicgatecell, subgate1, sub1convergence, c3c4_activsub1, sub1outputs,
    #                  subgate2,  sub2convergence, c3c4_activsub2, sub2outputs,
    #                  cellconvergence, c5_activ, summedoutputs]
    for i in range(len(_networkinfo)):
        _excelinfo[1].write(_row, i, str(_networkinfo[i]))

    allsuccess = True
    allfailed = True
    for pattern in range(len(_results)):
        if _results[pattern][9] == None:
            allsuccess = False
        else:
            allfailed = False
    if allsuccess:
        _excelinfo[1].write(_row, len(_networkinfo), "success")
    elif allfailed:
        _excelinfo[1].write(_row, len(_networkinfo), "failure")
    else:
        _excelinfo[1].write(_row, len(_networkinfo), "partial")

    for j in range(len(_results)):
        subrow = ((_row-1) * len(_results)) + j + 1
        for i in range(len(_networkinfo)):
            _excelinfo[2].write(subrow, i, str(_networkinfo[i]))
        for k in range(len(_results[j])):
            _excelinfo[2].write(subrow, len(_networkinfo) + k, str(_results[j][k]))

"""
Loops through the program for each cellsize, sparsity, inputsparsity and pattern.
Initializes a nodenetwork and spreads activation.
Stores the results of the learning function into a list and forwards this to the write function.
"""
cellsize        = [50] # 100
intsparsity     = [.98]
extsparsity     = [1, .95, .90, .85, .50, .20]
conn_strength   = [0.2, 0.5, 0.8, 0.0] # 0.0 for random
seeds           = [ [1,2,3,4],
                    [5,6,7,8] ]
patterns        = ["AND", "NAND", "OR", "NOR", "XOR", "XNOR"]
resultscounter  = 409

for size in range(len(cellsize)):
    for intspars in range(len(intsparsity)):
        for extspars in range(len(extsparsity)):
            for strength in range(len(conn_strength)):
                print("Current iteration: " + str(resultscounter))
                networkinfo = [resultscounter, cellsize[size],
                               intsparsity[intspars], extsparsity[extspars], conn_strength[strength]]
```

```
                print("Cellnetwork with nodes=" +
   str(cellsize[size]) + ", intspars=" +
                      str(intsparsity[intspars]) + ", extspars=" +
   str(extsparsity[extspars]) +
                      ", conn_str=" + str(conn_strength[strength]))
                results = []
                for p in range(len(patterns)):
                    print("Logic gate: " + patterns[p])
                    cn = CellNetwork(cellsize[size],
   intsparsity[intspars], extsparsity[extspars],
                                     seeds,
   conn_strength[strength])
                    results.append(cn.main(patterns[p]))
                addresultstoexcel([wb, sh1, sh2], networkinfo,
   results, resultscounter)
                wb.save("results_v5.72_cs50_is98.xls")
                print("Saved results to Excel")
                resultscounter += 1
```

Subcell

```
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 28 12:08:19 2020
Adaptation: 30-06-2020

@author: Kevin Liu & Robbin Koopman
"""

import numpy as np
 import random
 import math

 class Subcell():

     def __init__(self, _cellnodes, _seedcombo, _intconnspars,
   _extconnspars, _connstr, _timesteps = 10, _inputnodes = 2):
         outputnodes = 1 # Amount of output nodes
         self.skip = -99

         # array for storing all activation levels of all cellnodes
   at each timestep
         # for each inputpattern
         self.activation_cellnodes = np.zeros(4 * _timesteps *
   _cellnodes).reshape(
                                             4, _timesteps,
   _cellnodes)

         # Generating connection arrays
         self.inputconnections = self.networkgen(_inputnodes,
   _cellnodes, _seedcombo[0], _intconnspars, _connstr)
         self.nodeconnections = self.networkgen(_cellnodes,
   _cellnodes, _seedcombo[1], _intconnspars, _cellmidlayer=True)
         self.outputconnections = self.networkgen(_cellnodes,
   outputnodes, _seedcombo[2],  _intconnspars, _connstr)
         self.intercellconnections = self.networkgen(_cellnodes,
   _cellnodes, _seedcombo[3], _extconnspars, _connstr)

         # Values for control;
```

```
31.           # controlactivation adds activation as additional input to
   all cellnodes
32.           # controlsignconnection inverts input activation
33.           self.controlactivation = 0.0
34.           self.controlsignconnection = 1.0
35.
36.       """
37.       Generates a network consisting of connections between
   _startnodes and _endnodes.
38.       Each _startnode can be connected to any number of _endnodes.
39.       Connectionstrength is the same for all connections; this can be
   changed later if needed.
40.       self.skip indicates no connection is made between _startnode[i]
   and _endnode[j].
41.       _cellmidlayer is only True when connecting cellnodes, as
   cellnodes should not be connected to themselves.
42.       In all other cases, it is false; e.g. inputnode 0 could be
   connected to cellnode 0.
43.
44.       Returns the generated network.
45.       """
46.       def networkgen(self, _startnodes, _endnodes, _seed, _threshold,
   _connstrength = 0, _cellmidlayer = False):
47.           dummyconnections = np.zeros(_startnodes *
   _endnodes).reshape(_startnodes, _endnodes)
48.           random.seed(_seed)
49.           for i in range(_startnodes):
50.               for j in range(_endnodes):
51.                   dummyconnections[i,j] = self.skip
52.                   if _cellmidlayer and j == i:
53.                           continue
54.                   if random.random() >= _threshold:
55.                       if not _connstrength == 0.0:
56.                           dummyconnections[i, j] = _connstrength
57.                       else:
58.                           dummyconnections[i, j] =
   round(random.random(),4)
59.           return dummyconnections
60.
61.       """
62.       Calculates the total activation onto cellnode q, given by:
63.       - the inputnode * controlsignconnection
64.       - activation of cellnodes k at timestep t connected to cellnode
   q * connectionstrength of k to q
65.       - activation of othercellnodes k at timestep t connected to
   cellnode q * connectionstrength of k to q
66.       Stores the squashed inputSum as activation of cellnode q in
   timestep t+1.
67.       For the last timestep, also apply controlactivation before
   squashing.
68.       Lastly, store the end activations into the activationperinput
   array (used by the getoutput function).
69.       """
70.       def spreadactivation(self, _timestep, _input, _inputnumber,
   _othercell=None, _controlinsquash=True):
71.           if _othercell != None:
72.               inputothercell =
   _othercell.activation_cellnodes[_inputnumber, _timestep]
73.           nextstep = _timestep + 1
74.           for q in range(len(self.activation_cellnodes[_inputnumber,
   _timestep])):
```

```
75.             inputsum = 0.0
76.             for k in range(len(_input)):
77.                 if self.inputconnections[k, q] != self.skip:
78.                     inputsum += _input[k] *
   self.inputconnections[k, q] * self.controlsignconnection
79.             for k in
   range(len(self.activation_cellnodes[_inputnumber, _timestep])):
80.                 if self.nodeconnections[k, q] != self.skip:
81.                     inputsum +=
   self.activation_cellnodes[_inputnumber, _timestep, k] *
   self.nodeconnections[k, q]
82.             if _othercell != None:
83.                 for k in range(len(inputothercell)):
84.                     if _othercell.intercellconnections[k, q] !=
   self.skip:
85.                         inputsum += inputothercell[k] *
   _othercell.intercellconnections[k, q]
86.             if _controlinsquash:
87.                 if nextstep ==
   (len(self.activation_cellnodes[_inputnumber])-1):
88.                     inputsum += self.controlactivation
89.
90.             self.activation_cellnodes[_inputnumber, nextstep, q] =
   round(math.tanh(inputsum),4)
91.
92.     """
93.     Calculates the output of the subcell;
94.     Takes the activations of each cellnode at the last timestep
   (stored in self.activationperinput),
95.     multiplies each activation by the connectionstrength of
   cellnode q on outputnode,
96.     and adds all the (activations*connectionstrengths) together.
97.
98.     Returns the sum of all (activations*connectionstrengths)
99.     """
100.    def getoutput(self, _inputnumber):
101.        outputsum = 0.0
102.        for k in range(len(self.activation_cellnodes[_inputnumber,
   -1])):
103.            if self.outputconnections[k, 0] != self.skip:
104.                outputsum +=
   self.activation_cellnodes[_inputnumber, -1, k] *
   self.outputconnections[k, 0]
105.        return round(outputsum, 4)
106.
107.    """
108.    Solves the intermediate logic gates.
109.    Requires the inputpattern, pattern to be solved and the
   learningrate.
110.    Adjusts the input signconnection if input (0,0) expects an
   output > 0.
111.    Checks for error in the outputpattern and adjusts
   controlactication accordingly.
112.    Returns True when the subcell converged, the newly adjusted
   controlactication, and the outputs of the subcell.
113.    """
114.    def learn_subpattern(self, _input, _desiredoutput,
   _learningrate):
115.        learned = True
116.        errvalue = 0
117.        summedoutputs = np.zeros(len(_desiredoutput))
```

```
        outputpattern = np.zeros(len(_desiredoutput))

        for i in range(len(_input)):
            # Check if input = [0,0] desires output > 0. If yes, case must be NAND/NOR, thus reversesignactivation is called.
            if (_input[i] == np.array([0,0])).all():
                if _desiredoutput[i] > 0:
                    self.reversecontrolsignconnection()

            summedoutputs[i] = self.getoutput(i)
            if summedoutputs[i] > 0:
                outputpattern[i] = 1
            else:
                outputpattern[i] = 0

        # Check for error in outputpattern
        for i in range(len(_desiredoutput)):
            error = _desiredoutput[i] - outputpattern[i]
            if error != 0:
                learned = False
                errvalue = error

        # Apply learningrate to cellnodes
        if errvalue != 0:
            c3c4_activation = _learningrate * errvalue
            self.adjustcontrolactivation(c3c4_activation)

        return [learned, self.controlactivation, summedoutputs]

    """
    These functions represent the control nodes of the subcell;
    adjustcontrolactivation adds (error margin * learningrate) provided by the learning function to self.controlactivation.
    applycontrolactivation is only used for applying control after squashing. Should be called after spreading activation.
    reversecontrolsignconnection sets self.controlsignconnection to -1, thus inverting the input given to the cell only once.
    """
    def adjustcontrolactivation(self, _adjustment):
        self.controlactivation = round(self.controlactivation + _adjustment,4)
    def applycontrolactivation(self, _inputnumber):
        for c in range(len(self.activationperinput[_inputnumber])):
            self.activation_cellnodes[_inputnumber, -1, c] += self.controlactivation
    def reversecontrolsignconnection(self):
        self.controlsignconnection = -1
```

B4. Control nodes with separate input streams

CellNetwork

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jun 10 13:17:22 2020

@author: Kevin Liu & Robbin Koopman
"""

from Subcell import Subcell
```

```
9. import numpy as np
10.  import xlwt
11.
12.  class CellNetwork():
13.
14.      def __init__(self, _cellsize, _intspars, _extspars,
   _seedcombos, _connectionstrength):
15.          self.timesteps = 10
16.          self.learning_rate = 0.01
17.          self.controlactivationcell = 0.0
18.
19.          self.subcell1 = []
20.          self.subcell2 = []
21.          for c in range(2):
22.              b1 = Subcell(_cellsize, _seedcombos[c], self.timesteps,
   _intspars, _extspars, _connectionstrength)
23.              b2 = Subcell(_cellsize, _seedcombos[c+2],
   self.timesteps, _intspars, _extspars, _connectionstrength)
24.              self.subcell1.append(b1)
25.              self.subcell2.append(b2)
26.
27.      def main(self, _patternname):
28.          learningruns = 1000
29.          learningrate = 0.01
30.          despatterns = self.get_patterns(_patternname)
31.          inputpatterns = np.array(  [    [ 1, 1 ],
32.                                         [ 1, 0 ],
33.                                         [ 0, 1 ],
34.                                         [ 0, 0 ] ] )
35.          results = []
36.          sub1_results = []
37.          sub2_results = []
38.          sub1learned = sub2learned = False
39.
40.          for q in range(learningruns):
41.              errvalue = 0.0
42.              learned = True
43.              summedoutputs = np.zeros(len(despatterns[0]))
44.              outputpattern = np.zeros(len(despatterns[0]))
45.
46.              for i in range(len(inputpatterns)):
47.                  for t in range(self.timesteps-1):
48.                      if not sub1learned:
49.
   self.subcell1[0].spreadactivation(inputpatterns[i][0], t,
   self.subcell1[1], i)
50.
   self.subcell1[1].spreadactivation(inputpatterns[i][1], t,
   self.subcell1[0], i)
51.                      if not sub2learned:
52.
   self.subcell2[0].spreadactivation(inputpatterns[i][0], t,
   self.subcell2[1], i)
53.
   self.subcell2[1].spreadactivation(inputpatterns[i][1], t,
   self.subcell2[0], i)
54.
55.              if not sub1learned:
56.                  sub1_results = self.learn_subpattern(inputpatterns,
   despatterns[1], self.subcell1, learningrate)
57.                  sub1learned = sub1_results[0]
```

```
58.                 if sub1learned:
59.                     sub1_results[0] = q
60.             if not sub2learned:
61.                 sub2_results = self.learn_subpattern(inputpatterns,
    despatterns[2], self.subcell2, learningrate)
62.                 sub2learned = sub2_results[0]
63.                 if sub2learned:
64.                     sub2_results[0] = q
65.
66.
67.             for i in range(len(despatterns[0])):
68.                 summedoutputs[i] = (sub1_results[2][i] +
    sub2_results[2][i] +
69.                                     self.controlactivationcell)
70.                 if summedoutputs[i] > 0:
71.                     outputpattern[i] = 1
72.
73.             for i in range(len(despatterns[0])):
74.                 error = despatterns[0][i] - outputpattern[i]
75.                 if error != 0:
76.                     errvalue = error
77.                     learned = False
78.
79.             if errvalue != 0:
80.                 self.controlactivationcell =
    round(self.controlactivationcell + learningrate * errvalue, 4)
81.
82.             if learned:
83.                 print("Network learned after " + str(q) + "
    learningruns.")
84.                 learned = q
85.                 break
86.             elif q == learningruns-1:
87.                 print("Network failed to converge.")
88.                 learned = None
89.
90.         results.append(_patternname)
91.         results.append(despatterns[1])
92.         for r in range(len(sub1_results)):
93.             results.append(sub1_results[r])
94.         results.append(despatterns[2])
95.         for r2 in range(len(sub2_results)):
96.             results.append(sub2_results[r2])
97.
98.         results.append(learned)
99.         results.append(self.controlactivationcell)
100.        results.append(summedoutputs)
101.
102.        return results
103.
104.    def learn_subpattern(self, _input, _desiredoutput, _subcell,
    _learningrate = 0.01):
105.        errvalue = 0.0
106.        # print("Solving for subcell with desiredoutput " +
    str(_desiredoutput))
107.        outputpattern = np.zeros(len(_desiredoutput))
108.        summedoutputs = np.zeros(len(_desiredoutput))
109.
110.        for i in range(len(_input)):
111.            # Check if input = [0,0] desires output > 0. If yes,
    case must be NAND/NOR, thus reversesignactivation is called.
```

```
112.             if (_input[i] == np.array([0,0])).all():
113.                 if _desiredoutput[i] > 0:
114.                     for c in range(len(_subcell)):
115.                         _subcell[c].reversecontrolsignconnection()
116.
117.             summedoutputs[i] = 0.0
118.             for c in range(len(_subcell)):
119.                 summedoutputs[i] += _subcell[c].getoutput(i)
120.             # print("Summedoutputs subcell@" + str(_input[i]) + ':
    ' + str(summedoutput))
121.             if summedoutputs[i] > 0:
122.                 outputpattern[i] = 1
123.             else:
124.                 outputpattern[i] = 0
125.
126.         # Check for error in outputpattern
127.         for i in range(len(_desiredoutput)):
128.             error = _desiredoutput[i] - outputpattern[i]
129.             if error != 0:
130.                 errvalue = error
131.
132.         # Apply learningrate to cellnodes
133.         if errvalue != 0:
134.             c3c4_activation = _learningrate * errvalue
135.             # print("Adjusting c3c4 activation in subcell with
    desiredpattern " + str(_desiredoutput))
136.             for c in range(len(_subcell)):
137.
    _subcell[c].adjustcontrolactivation(c3c4_activation)
138.             # print(" ")
139.             return [False, _subcell[0].controlactivation,
    summedoutputs]
140.         else:
141.             # print("No error in subcell with desiredpattern " +
    str(_desiredoutput))
142.             # print(" ")
143.             return [True, _subcell[0].controlactivation,
    summedoutputs]
144.
145.     def get_patterns(self, _patternname):
146.         if _patternname == "AND":
147.             desiredpattern = [1,0,0,0]
148.             subpattern1 = [1,0,0,0]
149.             subpattern2 = [1,0,0,0]
150.         elif _patternname == "OR":
151.             desiredpattern = [1,1,1,0]
152.             subpattern1 = [1,1,1,0]
153.             subpattern2 = [1,1,1,0]
154.         elif _patternname == "NOR":
155.             desiredpattern = [0,0,0,1]
156.             subpattern1 = [0,0,0,1]
157.             subpattern2 = [0,0,0,1]
158.         elif _patternname == "NAND":
159.             desiredpattern = [0,1,1,1]
160.             subpattern1 = [0,1,1,1]
161.             subpattern2 = [0,1,1,1]
162.         elif _patternname == "XOR":
163.             desiredpattern = [0,1,1,0]
164.             subpattern1 = [1,1,1,0]
165.             subpattern2 = [0,1,1,1]
166.         elif _patternname == "XNOR":
```

```
167.             desiredpattern = [1,0,0,1]
168.             subpattern1 = [1,0,0,0]
169.             subpattern2 = [0,0,0,1]
170.
171.         return [desiredpattern, subpattern1, subpattern2]
172.
173. wb = xlwt.Workbook()
174. sh1 = wb.add_sheet("successrate")
175. sh1.write(0,0,"id")
176. sh1.write(0,1,"cellsize")
177. sh1.write(0,2,"intsparsity")
178. sh1.write(0,3,"extsparsity")
179. sh1.write(0,4,"conn_strength")
180. sh1.write(0,5,"result")
181.
182. sh2 = wb.add_sheet("logicgates")
183. sh2.write(0,0,"id")
184. sh2.write(0,1,"cellsize")
185. sh2.write(0,2,"intsparsity")
186. sh2.write(0,3,"extsparsity")
187. sh2.write(0,4,"conn_strength")
188. sh2.write(0,5,"logicgate_cell")
189. sh2.write(0,6,"subgate1")
190. sh2.write(0,7,"sub1convergence")
191. sh2.write(0,8,"c3c4_activsub1")
192. sh2.write(0,9,"sub1outputs")
193. sh2.write(0,10,"subgate2")
194. sh2.write(0,11,"sub2convergence")
195. sh2.write(0,12,"c3c4_activsub2")
196. sh2.write(0,13,"sub2outputs")
197. sh2.write(0,14,"cellconvergence")
198. sh2.write(0,15,"c5_activ")
199. sh2.write(0,16,"summedoutputs")
200.
201. resultscounter = 1
202.
203. """
204. Writes the results of the learn function into an excel file.
205. Sheet 1 shows the success vs partial vs failure rates of the cell
   with varying size, sparsity, conn_strength and seed
206. Sheet 2 contains detailed information per logic gate; learningruns
   for reaching convergence, bias and output activations
207. """
208. def addresultstoexcel(_excelinfo, _networkinfo, _results, _row):
209.     # _excelinfo    = [workbook, sheet1, sheet2]
210.     # _networkinfo  = [cellsize, sparsity, conn_str, seeds]
211.     # _results      = [logicgatecell, subgate1, sub1convergence,
   c3c4_activsub1, sub1outputs,
212.     #                  subgate2,  sub2convergence, c3c4_activsub2,
   sub2outputs,
213.     #                  cellconvergence, c5_activ, summedoutputs]
214.     for i in range(len(_networkinfo)):
215.         _excelinfo[1].write(_row, i, str(_networkinfo[i]))
216.
217.     allsuccess = True
218.     allfailed = True
219.     for pattern in range(len(_results)):
220.         if _results[pattern][9] == None:
221.             allsuccess = False
222.         else:
223.             allfailed = False
```

```
224.     if allsuccess:
225.         _excelinfo[1].write(_row, len(_networkinfo), "success")
226.     elif allfailed:
227.         _excelinfo[1].write(_row, len(_networkinfo), "failure")
228.     else:
229.         _excelinfo[1].write(_row, len(_networkinfo), "partial")
230.
231.     for j in range(len(_results)):
232.         subrow = ((_row-1) * len(_results)) + j + 1
233.         for i in range(len(_networkinfo)):
234.             _excelinfo[2].write(subrow, i, str(_networkinfo[i]))
235.         for k in range(len(_results[j])):
236.             _excelinfo[2].write(subrow, len(_networkinfo) + k,
    str(_results[j][k]))
237.
238.
239. """
240. Loops through the program for each cellsize, sparsity,
    inputsparsity and pattern.
241. Initializes a nodenetwork and spreads activation.
242. Stores the results of the learning function into a list and
    forwards this to the write function.
243. """
244. cellsize = [10, 20, 50, 100]
245. sparsity = [.80, .85, .88, .92, .95, .98]
246. externalsparsity = [1, .95, .90, .85, .50, .20]
247. conn_strength = [0.2, 0.5, 0.8, 0.0] # 0.0 for random
248. seeds = [ [1,2,3,4],
249.           [5,6,7,8],
250.           [9,10,11,12],
251.           [13,14,15,16] ]
252. patterns = ["AND", "OR", "NAND", "NOR", "XOR", "XNOR"]
253.
254. for size in range(len(cellsize)):
255.     for intspars in range(len(sparsity)):
256.         for extspars in range(len(externalsparsity)):
257.             for strength in range(len(conn_strength)):
258.                 print("Current iteration: " + str(resultscounter))
259.                 networkinfo = [resultscounter,cellsize[size],
    sparsity[intspars], externalsparsity[extspars],
    conn_strength[strength]]
260.                 print("Cellnetwork with nodes=" +
    str(cellsize[size]) + ", intspars=" +
261.                       str(sparsity[intspars]) + ", extspars=" +
    str(externalsparsity[extspars]) +
262.                       ", conn_str=" + str(conn_strength[strength]))
263.                 results = []
264.                 for p in range(len(patterns)):
265.                     print("Logic gate: " + patterns[p])
266.                     cn = CellNetwork(cellsize[size],
    sparsity[intspars], externalsparsity[extspars], seeds,
    conn_strength[strength])
267.                     results.append(cn.main(patterns[p]))
268.
269.                 addresultstoexcel([wb, sh1, sh2], networkinfo,
    results, resultscounter)
270.                 wb.save("results_v5.11.xls")
271.                 print("Saved results to Excel")
272.                 resultscounter += 1
```

Subcell

```
1. # -*- coding: utf-8 -*-
2. """
3. Created on Tue Apr 28 12:08:19 2020
4. Adaptation: 30-06-2020
5.
6. @author: Kevin Liu & Robbin Koopman
7. """
8.
9. import numpy as np
10.  import random
11.  import math
12.
13.  class Subcell():
14.
15.      def __init__(self, _cellnodes, _seedcombo, _timesteps,
   _intconnspars, _extconnspars, _connstr):
16.          inputnodes = 1 # Amount of input nodes
17.          outputnodes = 1 # Amount of output nodes
18.          self.skip = -99
19.
20.          # array for storing all activation levels of all cellnodes
   at each timestep
21.          # for each inputpattern
22.          self.activation_cellnodes = np.zeros(4 * _timesteps *
   _cellnodes).reshape(
23.                                               4, _timesteps,
   _cellnodes)
24.
25.          # Generating connection arrays
26.          self.inputconnections = self.networkgen(inputnodes,
   _cellnodes, _seedcombo[0], _intconnspars, _connstr)
27.          self.nodeconnections = self.networkgen(_cellnodes,
   _cellnodes, _seedcombo[1], _intconnspars, _cellmidlayer=True)
28.          self.outputconnections = self.networkgen(_cellnodes,
   outputnodes, _seedcombo[2],  _intconnspars, _connstr)
29.          self.intercellconnections = self.networkgen(_cellnodes,
   _cellnodes, _seedcombo[3], _extconnspars, _connstr)
30.
31.          # Values for control;
32.          # controlactivation adds activation as additional input to
   all cellnodes
33.          # controlsignconnection inverts input activation
34.          self.controlactivation = 0.0
35.          self.controlsignconnection = 1.0
36.
37.      """
38.      Generates a network consisting of connections between
   _startnodes and _endnodes.
39.      Each _startnode can be connected to any number of _endnodes.
40.      Connectionstrength is the same for all connections; this can be
   changed later if needed.
41.      self.skip indicates no connection is made between _startnode[i]
   and _endnode[j].
42.      _cellmidlayer is only True when connecting cellnodes, as
   cellnodes should not be connected to themselves.
43.      In all other cases, it is false; e.g. inputnode 0 could be
   connected to cellnode 0.
44.
```

```
45.       Returns the generated network.
46.       """
47.       def networkgen(self, _startnodes, _endnodes, _seed, _threshold,
   _connstrength = 0, _cellmidlayer = False):
48.           dummyconnections = np.zeros(_startnodes *
   _endnodes).reshape(_startnodes, _endnodes)
49.           random.seed(_seed)
50.           for i in range(_startnodes):
51.               for j in range(_endnodes):
52.                   dummyconnections[i,j] = self.skip
53.                   if _cellmidlayer and j == i:
54.                           continue
55.                   if random.random() >= _threshold:
56.                       if not _connstrength == 0.0:
57.                           dummyconnections[i, j] = _connstrength
58.                       else:
59.                           dummyconnections[i, j] =
   round(random.random(),4)
60.           return dummyconnections
61.
62.       """
63.       Calculates the total activation onto cellnode q, given by:
64.       - the inputnode * controlsignconnection
65.       - activation of cellnodes k at timestep t connected to cellnode
   q * connectionstrength of k to q
66.       - activation of othercellnodes k at timestep t connected to
   cellnode q * connectionstrength of k to q
67.       Stores the squashed inputSum as activation of cellnode q in
   timestep t+1.
68.       For the last timestep, also apply controlactivation before
   squashing.
69.       Lastly, store the end activations into the activationperinput
   array (used by the getoutput function).
70.       """
71.       def spreadactivation(self, _input, _timestep, _othercell,
   _inputnumber, _controlinsquash = True):
72.           if _othercell != None:
73.               inputothercell =
   _othercell.activation_cellnodes[_inputnumber, _timestep]
74.           nextstep = _timestep + 1
75.           for q in range(len(self.activation_cellnodes[_inputnumber,
   _timestep])):
76.               inputsum = 0.0
77.               if self.inputconnections[0, q] != self.skip:
78.                   inputsum = _input * self.inputconnections[0, q] *
   self.controlsignconnection
79.               for k in
   range(len(self.activation_cellnodes[_inputnumber, _timestep])):
80.                   if self.nodeconnections[k, q] != self.skip:
81.                       inputsum +=
   self.activation_cellnodes[_inputnumber, _timestep, k] *
   self.nodeconnections[k, q]
82.               if _othercell != None:
83.                   for k in range(len(inputothercell)):
84.                       if _othercell.intercellconnections[k, q] !=
   self.skip:
85.                           inputsum += inputothercell[k] *
   _othercell.intercellconnections[k, q]
86.               if _controlinsquash:
87.                   if nextstep ==
   (len(self.activation_cellnodes[_inputnumber])-1):
```

```
                    inputsum += self.controlactivation

            self.activation_cellnodes[_inputnumber, nextstep, q] =
round(math.tanh(inputsum),4)

    """
    Calculates the output of the subcell;
    Takes the activations of each cellnode at the last timestep
(stored in self.activationperinput),
    multiplies each activation by the connectionstrength of
cellnode q on outputnode,
    and adds all the (activations*connectionstrengths) together.

    Returns the sum of all (activations*connectionstrengths)
    """
    def getoutput(self, _inputnumber):
        outputsum = 0.0
        for k in range(len(self.activation_cellnodes[_inputnumber,
-1])):
            if self.outputconnections[k, 0] != self.skip:
                outputsum +=
self.activation_cellnodes[_inputnumber, -1, k] *
self.outputconnections[k, 0]
        return round(outputsum, 4)

    """
    These functions represent the control nodes of the subcell;
    adjustcontrolactivation adds (error margin * learningrate)
provided by the learning function to self.controlactivation.
    applycontrolactivation is only used for applying control after
squashing. Should be called after spreading activation.
    reversecontrolsignconnection sets self.controlsignconnection to
-1, thus inverting the input given to the cell only once.
    """
    def adjustcontrolactivation(self, _adjustment):
        self.controlactivation = round(self.controlactivation +
_adjustment,4)
    def applycontrolactivation(self, _inputnumber):
        for c in range(len(self.activationperinput[_inputnumber])):
            self.activation_cellnodes[_inputnumber, -1, c] +=
self.controlactivation
    def reversecontrolsignconnection(self):
        self.controlsignconnection = -1
```

B5. Control nodes and saturation cells

CellNetwork

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jun 10 13:17:22 2020

@author: Kevin Liu & Robbin Koopman
"""

from Subcell import Subcell
import numpy as np
 import xlwt

 class CellNetwork():
```

```
13.
14.      def __init__(self, _cellsize, _intspars, _extspars,
   _seedcombos, _connectionstrength):
15.          self.timesteps = 10
16.          self.learning_rate = 0.01
17.          self.controlactivationcell = 0.0
18.
19.          self.subcell1 = []
20.          self.subcell2 = []
21.          for c in range(2):
22.              b1 = Subcell(_cellsize, _seedcombos[c], self.timesteps,
   _intspars, _extspars, _connectionstrength)
23.              b2 = Subcell(_cellsize, _seedcombos[c+2],
   self.timesteps, _intspars, _extspars, _connectionstrength)
24.              self.subcell1.append(b1)
25.              self.subcell2.append(b2)
26.
27.          self.saturationcell1 = Subcell(20, [0,0,0,0],
   self.timesteps, 0, 0, 0.01)
28.          self.saturationcell2 = Subcell(20, [0,0,0,0],
   self.timesteps, 0, 0, 0.01)
29.
30.      def main(self, _patternname):
31.          learningruns = 1000
32.          learningrate = 0.01
33.          despatterns = self.get_patterns(_patternname)
34.          inputpatterns = np.array(  [     [ 1, 1 ],
35.                                           [ 1, 0 ],
36.                                           [ 0, 1 ],
37.                                           [ 0, 0 ] ] )
38.          results = []
39.          sub1_results = []
40.          sub2_results = []
41.          sub1learned = sub2learned = False
42.
43.          for q in range(learningruns):
44.              errvalue = 0.0
45.              learned = True
46.              summedoutputs = np.zeros(len(despatterns[0]))
47.              outputpattern = np.zeros(len(despatterns[0]))
48.
49.              for i in range(len(inputpatterns)):
50.                  for t in range(self.timesteps-1):
51.                      if not sub1learned:
52.
   self.subcell1[0].spreadactivation(inputpatterns[i][0], t,
   self.subcell1[1], i)
53.
   self.subcell1[1].spreadactivation(inputpatterns[i][1], t,
   self.subcell1[0], i)
54.                      if not sub2learned:
55.
   self.subcell2[0].spreadactivation(inputpatterns[i][0], t,
   self.subcell2[1], i)
56.
   self.subcell2[1].spreadactivation(inputpatterns[i][1], t,
   self.subcell2[0], i)
57.
58.              if not sub1learned:
59.                  sub1_results = self.learn_subpattern(inputpatterns,
   despatterns[1], self.subcell1, learningrate)
```

```
                    sub1learned = sub1_results[0]
                    if sub1learned:
                        sub1_results[0] = q
                if not sub2learned:
                    sub2_results = self.learn_subpattern(inputpatterns,
despatterns[2], self.subcell2, learningrate)
                    sub2learned = sub2_results[0]
                    if sub2learned:
                        sub2_results[0] = q

                for i in range(len(despatterns[0])):
                    for t in range(self.timesteps-1):

self.saturationcell1.spreadactivation(self.subcell1[0].getoutput(i)+s
elf.subcell1[1].getoutput(i), t, None, i, False)

self.saturationcell2.spreadactivation(self.subcell2[0].getoutput(i)+s
elf.subcell2[1].getoutput(i), t, None, i, False)

                    summedoutputs[i] =
(self.saturationcell1.getoutput(i) +
self.saturationcell2.getoutput(i) +
                                            self.controlactivationcell)

                    if summedoutputs[i] > 0:
                        outputpattern[i] = 1

                for i in range(len(despatterns[0])):
                    error = despatterns[0][i] - outputpattern[i]
                    if error != 0:
                        errvalue = error
                        learned = False

                if errvalue != 0:
                    self.controlactivationcell =
round(self.controlactivationcell + learningrate * errvalue, 4)

                if learned:
                    print("Network learned after " + str(q) + "
learningruns.")
                    learned = q
                    break
                elif q == learningruns-1:
                    print("Network failed to converge.")
                    learned = None

            results.append(_patternname)
            results.append(despatterns[1])
            for r in range(len(sub1_results)):
                results.append(sub1_results[r])
            results.append(despatterns[2])
            for r2 in range(len(sub2_results)):
                results.append(sub2_results[r2])

            results.append(learned)
            results.append(self.controlactivationcell)
            results.append(summedoutputs)

            return results

```

```
111.     def learn_subpattern(self, _input, _desiredoutput, _subcell,
   _learningrate = 0.01):
112.         errvalue = 0.0
113.         # print("Solving for subcell with desiredoutput " +
   str(_desiredoutput))
114.         outputpattern = np.zeros(len(_desiredoutput))
115.         summedoutputs = np.zeros(len(_desiredoutput))
116.
117.         for i in range(len(_input)):
118.             # Check if input = [0,0] desires output > 0. If yes,
   case must be NAND/NOR, thus reversesignactivation is called.
119.             if (_input[i] == np.array([0,0])).all():
120.                 if _desiredoutput[i] > 0:
121.                     for c in range(len(_subcell)):
122.                         _subcell[c].reversecontrolsignconnection()
123.
124.             summedoutputs[i] = 0.0
125.             for c in range(len(_subcell)):
126.                 summedoutputs[i] += _subcell[c].getoutput(i)
127.             # print("Summedoutputs subcell@" + str(_input[i]) + ':
   ' + str(summedoutput))
128.             if summedoutputs[i] > 0:
129.                 outputpattern[i] = 1
130.             else:
131.                 outputpattern[i] = 0
132.
133.         # Check for error in outputpattern
134.         for i in range(len(_desiredoutput)):
135.             error = _desiredoutput[i] - outputpattern[i]
136.             if error != 0:
137.                 errvalue = error
138.
139.         # Apply learningrate to cellnodes
140.         if errvalue != 0:
141.             c3c4_activation = _learningrate * errvalue
142.             # print("Adjusting c3c4 activation in subcell with
   desiredpattern " + str(_desiredoutput))
143.             for c in range(len(_subcell)):
144.
   _subcell[c].adjustcontrolactivation(c3c4_activation)
145.             # print(" ")
146.             return [False, _subcell[0].controlactivation,
   summedoutputs]
147.         else:
148.             # print("No error in subcell with desiredpattern " +
   str(_desiredoutput))
149.             # print(" ")
150.             return [True, _subcell[0].controlactivation,
   summedoutputs]
151.
152.     def get_patterns(self, _patternname):
153.         if _patternname == "AND":
154.             desiredpattern = [1,0,0,0]
155.             subpattern1 = [1,0,0,0]
156.             subpattern2 = [1,0,0,0]
157.         elif _patternname == "OR":
158.             desiredpattern = [1,1,1,0]
159.             subpattern1 = [1,1,1,0]
160.             subpattern2 = [1,1,1,0]
161.         elif _patternname == "NOR":
162.             desiredpattern = [0,0,0,1]
```

```
163.             subpattern1 = [0,0,0,1]
164.             subpattern2 = [0,0,0,1]
165.         elif _patternname == "NAND":
166.             desiredpattern = [0,1,1,1]
167.             subpattern1 = [0,1,1,1]
168.             subpattern2 = [0,1,1,1]
169.         elif _patternname == "XOR":
170.             desiredpattern = [0,1,1,0]
171.             subpattern1 = [1,1,1,0]
172.             subpattern2 = [0,1,1,1]
173.         elif _patternname == "XNOR":
174.             desiredpattern = [1,0,0,1]
175.             subpattern1 = [1,0,0,0]
176.             subpattern2 = [0,0,0,1]
177.
178.         return [desiredpattern, subpattern1, subpattern2]
179.
180. wb = xlwt.Workbook()
181. sh1 = wb.add_sheet("successrate")
182. sh1.write(0,0,"id")
183. sh1.write(0,1,"cellsize")
184. sh1.write(0,2,"intsparsity")
185. sh1.write(0,3,"extsparsity")
186. sh1.write(0,4,"conn_strength")
187. sh1.write(0,5,"result")
188.
189. sh2 = wb.add_sheet("logicgates")
190. sh2.write(0,0,"id")
191. sh2.write(0,1,"cellsize")
192. sh2.write(0,2,"intsparsity")
193. sh2.write(0,3,"extsparsity")
194. sh2.write(0,4,"conn_strength")
195. sh2.write(0,5,"logicgate_cell")
196. sh2.write(0,6,"subgate1")
197. sh2.write(0,7,"sub1convergence")
198. sh2.write(0,8,"c3c4_activsub1")
199. sh2.write(0,9,"sub1outputs")
200. sh2.write(0,10,"subgate2")
201. sh2.write(0,11,"sub2convergence")
202. sh2.write(0,12,"c3c4_activsub2")
203. sh2.write(0,13,"sub2outputs")
204. sh2.write(0,14,"cellconvergence")
205. sh2.write(0,15,"c5_activ")
206. sh2.write(0,16,"summedoutputs")
207.
208. resultscounter = 277
209.
210. """
211. Writes the results of the learn function into an excel file.
212. Sheet 1 shows the success vs partial vs failure rates of the cell
     with varying size, sparsity, conn_strength and seed
213. Sheet 2 contains detailed information per logic gate; learningruns
     for reaching convergence, bias and output activations
214. """
215. def addresultstoexcel(_excelinfo, _networkinfo, _results, _row):
216.     # _excelinfo    = [workbook, sheet1, sheet2]
217.     # _networkinfo  = [cellsize, sparsity, conn_str, seeds]
218.     # _results      = [logicgatecell, subgate1, sub1convergence,
     c3c4_activsub1, sub1outputs,
219.     #                 subgate2,  sub2convergence, c3c4_activsub2,
     sub2outputs,
```

```
220.     #                 cellconvergence, c5_activ, summedoutputs]
221.     for i in range(len(_networkinfo)):
222.         _excelinfo[1].write(_row, i, str(_networkinfo[i]))
223.
224.     allsuccess = True
225.     allfailed = True
226.     for pattern in range(len(_results)):
227.         if _results[pattern][9] == None:
228.             allsuccess = False
229.         else:
230.             allfailed = False
231.     if allsuccess:
232.         _excelinfo[1].write(_row, len(_networkinfo), "success")
233.     elif allfailed:
234.         _excelinfo[1].write(_row, len(_networkinfo), "failure")
235.     else:
236.         _excelinfo[1].write(_row, len(_networkinfo), "partial")
237.
238.     for j in range(len(_results)):
239.         subrow = ((_row-1) * len(_results)) + j + 1
240.         for i in range(len(_networkinfo)):
241.             _excelinfo[2].write(subrow, i, str(_networkinfo[i]))
242.         for k in range(len(_results[j])):
243.             _excelinfo[2].write(subrow, len(_networkinfo) + k,
    str(_results[j][k]))
244.
245.
246. """
247. Loops through the program for each cellsize, sparsity,
    inputsparsity and pattern.
248. Initializes a nodenetwork and spreads activation.
249. Stores the results of the learning function into a list and
    forwards this to the write function.
250. """
251. cellsize = [20, 50] #10, 100
252. sparsity = [.80, .85, .88, .92, .95, .98]
253. externalsparsity = [1, .95, .90, .85, .50, .20]
254. conn_strength = [0.2, 0.5, 0.8, 0.0] # 0.0 for random
255. seeds = [ [1,2,3,4],
256.           [5,6,7,8],
257.           [9,10,11,12],
258.           [13,14,15,16] ]
259. patterns = ["AND", "OR", "NAND", "NOR", "XOR", "XNOR"]
260.
261. for size in range(len(cellsize)):
262.     for intspars in range(len(sparsity)):
263.         for extspars in range(len(externalsparsity)):
264.             for strength in range(len(conn_strength)):
265.                 print("Current iteration: " + str(resultscounter))
266.                 networkinfo = [resultscounter,cellsize[size],
    sparsity[intspars], externalsparsity[extspars],
    conn_strength[strength]]
267.                 print("Cellnetwork with nodes=" +
    str(cellsize[size]) + ", intspars=" +
268.                       str(sparsity[intspars]) + ", extspars=" +
    str(externalsparsity[extspars]) +
269.                       ", conn_str=" + str(conn_strength[strength]))
270.                 results = []
271.                 for p in range(len(patterns)):
272.                     print("Logic gate: " + patterns[p])
```

```
273.                     cn = CellNetwork(cellsize[size],
   sparsity[intspars], externalsparsity[extspars], seeds,
   conn_strength[strength])
274.                     results.append(cn.main(patterns[p]))
275.
276.                 addresultstoexcel([wb, sh1, sh2], networkinfo,
   results, resultscounter)
277.                 wb.save("results_v5.62_p3.xls")
278.                 print("Saved results to Excel")
279.                 resultscounter += 1
```

Subcell

```
1. # -*- coding: utf-8 -*-
2. """
3. Created on Tue Apr 28 12:08:19 2020
4. Adaptation: 30-06-2020
5.
6. @author: Kevin Liu & Robbin Koopman
7. """
8.
9. import numpy as np
10.  import random
11.  import math
12.
13.  class Subcell():
14.
15.      def __init__(self, _cellnodes, _seedcombo, _timesteps,
   _intconnspars, _extconnspars, _connstr):
16.          inputnodes = 1 # Amount of input nodes
17.          outputnodes = 1 # Amount of output nodes
18.          self.skip = -99
19.
20.          # array for storing all activation levels of all cellnodes
   at each timestep
21.          # for each inputpattern
22.          self.activation_cellnodes = np.zeros(4 * _timesteps *
   _cellnodes).reshape(
23.                                                  4, _timesteps,
   _cellnodes)
24.
25.          # Generating connection arrays
26.          self.inputconnections = self.networkgen(inputnodes,
   _cellnodes, _seedcombo[0], _intconnspars, _connstr)
27.          self.nodeconnections = self.networkgen(_cellnodes,
   _cellnodes, _seedcombo[1], _intconnspars, _cellmidlayer=True)
28.          self.outputconnections = self.networkgen(_cellnodes,
   outputnodes, _seedcombo[2],  _intconnspars, _connstr)
29.          self.intercellconnections = self.networkgen(_cellnodes,
   _cellnodes, _seedcombo[3], _extconnspars, _connstr)
30.
31.          # Values for control;
32.          # controlactivation adds activation as additional input to
   all cellnodes
33.          # controlsignconnection inverts input activation
34.          self.controlactivation = 0.0
35.          self.controlsignconnection = 1.0
36.
37.      """
```

```
38.       Generates a network consisting of connections between
   _startnodes and _endnodes.
39.       Each _startnode can be connected to any number of _endnodes.
40.       Connectionstrength is the same for all connections; this can be
   changed later if needed.
41.       self.skip indicates no connection is made between _startnode[i]
   and _endnode[j].
42.       _cellmidlayer is only True when connecting cellnodes, as
   cellnodes should not be connected to themselves.
43.       In all other cases, it is false; e.g. inputnode 0 could be
   connected to cellnode 0.
44.
45.       Returns the generated network.
46.       """
47.       def networkgen(self, _startnodes, _endnodes, _seed, _threshold,
   _connstrength = 0, _cellmidlayer = False):
48.           dummyconnections = np.zeros(_startnodes *
   _endnodes).reshape(_startnodes, _endnodes)
49.           random.seed(_seed)
50.           for i in range(_startnodes):
51.               for j in range(_endnodes):
52.                   dummyconnections[i,j] = self.skip
53.                   if _cellmidlayer and j == i:
54.                           continue
55.                   if random.random() >= _threshold:
56.                       if not _connstrength == 0.0:
57.                           dummyconnections[i, j] = _connstrength
58.                       else:
59.                           dummyconnections[i, j] =
   round(random.random(),4)
60.           return dummyconnections
61.
62.       """
63.       Calculates the total activation onto cellnode q, given by:
64.       - the inputnode * controlsignconnection
65.       - activation of cellnodes k at timestep t connected to cellnode
   q * connectionstrength of k to q
66.       - activation of othercellnodes k at timestep t connected to
   cellnode q * connectionstrength of k to q
67.       Stores the squashed inputSum as activation of cellnode q in
   timestep t+1.
68.       For the last timestep, also apply controlactivation before
   squashing.
69.       Lastly, store the end activations into the activationperinput
   array (used by the getoutput function).
70.       """
71.       def spreadactivation(self, _input, _timestep, _othercell,
   _inputnumber, _controlinsquash = True):
72.           if _othercell != None:
73.               inputothercell =
   _othercell.activation_cellnodes[_inputnumber, _timestep]
74.           nextstep = _timestep + 1
75.           for q in range(len(self.activation_cellnodes[_inputnumber,
   _timestep])):
76.               inputsum = 0.0
77.               if self.inputconnections[0, q] != self.skip:
78.                   inputsum = _input * self.inputconnections[0, q] *
   self.controlsignconnection
79.               for k in
   range(len(self.activation_cellnodes[_inputnumber, _timestep])):
80.                   if self.nodeconnections[k, q] != self.skip:
```

```
81.                         inputsum +=
   self.activation_cellnodes[_inputnumber, _timestep, k] *
   self.nodeconnections[k, q]
82.                 if _othercell != None:
83.                     for k in range(len(inputothercell)):
84.                         if _othercell.intercellconnections[k, q] !=
   self.skip:
85.                             inputsum += inputothercell[k] *
   _othercell.intercellconnections[k, q]
86.                 if _controlinsquash:
87.                     if nextstep ==
   (len(self.activation_cellnodes[_inputnumber])-1):
88.                         inputsum += self.controlactivation
89.
90.                 self.activation_cellnodes[_inputnumber, nextstep, q] =
   round(math.tanh(inputsum),4)
91.
92.         """
93.         Calculates the output of the subcell;
94.         Takes the activations of each cellnode at the last timestep
   (stored in self.activationperinput),
95.         multiplies each activation by the connectionstrength of
   cellnode q on outputnode,
96.         and adds all the (activations*connectionstrengths) together.
97.
98.         Returns the sum of all (activations*connectionstrengths)
99.         """
100.        def getoutput(self, _inputnumber):
101.            outputsum = 0.0
102.            for k in range(len(self.activation_cellnodes[_inputnumber,
   -1])):
103.                if self.outputconnections[k, 0] != self.skip:
104.                    outputsum +=
   self.activation_cellnodes[_inputnumber, -1, k] *
   self.outputconnections[k, 0]
105.            return round(outputsum, 4)
106.
107.        """
108.        These functions represent the control nodes of the subcell;
109.        adjustcontrolactivation adds (error margin * learningrate)
   provided by the learning function to self.controlactivation.
110.        applycontrolactivation is only used for applying control after
   squashing. Should be called after spreading activation.
111.        reversecontrolsignconnection sets self.controlsignconnection to
   -1, thus inverting the input given to the cell only once.
112.        """
113.        def adjustcontrolactivation(self, _adjustment):
114.            self.controlactivation = round(self.controlactivation +
   _adjustment,4)
115.        def applycontrolactivation(self, _inputnumber):
116.            for c in range(len(self.activationperinput[_inputnumber])):
117.                self.activation_cellnodes[_inputnumber, -1, c] +=
   self.controlactivation
118.        def reversecontrolsignconnection(self):
119.            self.controlsignconnection = -1
```

B6. Semantic Network

CellNetwork

```
1. # -*- coding: utf-8 -*-
2. """
3. Created on Wed Jun 10 13:17:22 2020
4. Latest adaptation: 21-12-2020
5. Versionnumber: 7.2
6.
7. @authors: Kevin Liu & Robbin Koopman
8. with help of Frank van der Velde
9. based on the boron dopant cell by Chen et al. (2020):
   https://doi.org/10.1038/s41586-019-1901-0
10.  """
11.
12.  from Subcell import Subcell
13.  import numpy as np
14.  import xlwt
15.
16.  class CellNetwork():
17.
18.      def __init__(self, _cellsize, _intspars, _extspars, _seeds,
   _connstr):
19.          items = ["robin", "canary", "sunfish", "salmon"]
20.          relations = ["has", "can", "is", "isa"]
21.          self.commands = []
22.          for i in range(len(items)):
23.              for r in range(len(relations)):
24.                  self.commands.append(items[i] + " " + relations[r])
25.
26.          self.attributes = {
27.              0:  "animal",
28.              1:  "bird",
29.              2:  "fish",
30.              3:  "move",
31.              4:  "skin",
32.              5:  "feathers",
33.              6:  "fly",
34.              7:  "wings",
35.              8:  "red",
36.              9:  "sing",
37.              10: "yellow",
38.              11: "swim",
39.              12: "gills",
40.              13: "scales"
41.              }
42.
43.          self.subcells = np.empty(shape=len(self.attributes),
   dtype=Subcell)
44.          for a in range(len(self.subcells)):
45.              for s in range (len(_seeds)-1):
46.                  _seeds[s] = (s+1) + (a * (len(_seeds)-1))
47.              print(_seeds)
48.              self.subcells[a] = Subcell(_cellsize, _seeds,
   _intspars, _extspars, _connstr, len(self.commands))
49.
50.      def get_patterns(self):
51.          itempatterns = [
52.              [1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0], # robin
53.              [1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0], # canary
```

```
54.             [1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1], # sunfish
55.             [1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1]  # salmon
56.             ]
57.
58.         relationpatterns = [
59.             [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1], # has
60.             [0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0], # can
61.             [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0], # is
62.             [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  # isa
63.             ]
64.
65.
66.         # inputpatterns is the input given to each cell (each
    representing their own attribute).
67.         # it stores the input to be given per cell per command,
68.         # and consists of 2 numbers (input for item and input for
    relation)
69.         # e.g. the input for attribute "animal" (index 0) consists
    of 16 * 2 inputs ("robin has" ... "salmon isa")
70.         inputpatterns =
    np.zeros(len(self.attributes)*len(itempatterns)*len(relationpatterns)
    *2).reshape(
71.
    len(self.attributes),len(itempatterns)*len(relationpatterns),2)
72.
73.         # desiredoutput is the output required from each cell
    (representing an attribute).
74.         # it stores the output desired per cell per command.
75.         # e.g. the desiredoutput for attribute "animal" (index 0)
    consists of 16 outputs ("robin has" ... "salmon isa")
76.         desiredoutput =
    np.zeros(len(self.attributes)*len(itempatterns)*len(relationpatterns)
    ).reshape(
77.
    len(self.attributes),len(itempatterns)*len(relationpatterns))
78.
79.         commandnum = 0
80.         for i in range(len(itempatterns)):
81.             for r in range(len(relationpatterns)):
82.                 for a in range(len(desiredoutput)):
83.                     inputpatterns[a][commandnum] =
    [itempatterns[i][a], relationpatterns[r][a]]
84.                     summedinput = itempatterns[i][a] +
    relationpatterns[r][a]
85.                     if summedinput == 2:
86.                         desiredoutput[a][commandnum] = 1
87.                 commandnum += 1
88.         return[inputpatterns, desiredoutput]
89.
90.
91.     def learn_pattern(self, _inputpatterns, _desiredoutputs):
92.         learningruns = 1000
93.         learningrate = 0.01
94.         timesteps = 10
95.
96.         attributeslearned = [None]*len(self.attributes)
97.         summedoutputs = np.zeros(len(self.subcells) *
    len(self.commands)).reshape(
98.             len(self.subcells), len(self.commands))
99.
100.        for q in range(learningruns):
```

```
101.             for attr in range(len(self.attributes)):
102.                 if attributeslearned[attr] != None:
103.                     continue
104.                 for comm in range(len(_inputpatterns[attr])):
105.                     for t in range(timesteps-1):
106.                         self.subcells[attr].spreadactivation(t,
    _inputpatterns[attr, comm], comm)
107.
108.                     summedoutputs[attr][comm] =
    self.subcells[attr].getoutput(comm)
109.
110.                 results =
    self.subcells[attr].learn_subpattern(_inputpatterns[attr],
    _desiredoutputs[attr], learningrate)
111.                 if results[0]:
112.                     attributeslearned[attr] = q
113.                     print("learned attribute " +
    str(self.get_attributes([attr])) + " at learningrun " + str(q))
114.
115.             # if all attributes learned, stop the learning
    function.
116.             if not any(x == None for x in attributeslearned):
117.                 break
118.
119.         # Checking if the resulting attributes per command are
    correct.
120.         commandslearned = [False]*len(self.commands)
121.         desired_attr_num = []
122.         returned_attr_num = []
123.
124.         desired_attr = []
125.         returned_attr = []
126.         for comm in range(len(self.commands)):
127.             desired_attr_num.append([])
128.             returned_attr_num.append([])
129.             for attr in range(len(self.attributes)):
130.                 if _desiredoutputs[attr][comm] > 0:
131.                     desired_attr_num[comm].append(attr)
132.                 if summedoutputs[attr][comm] > 0:
133.                     returned_attr_num[comm].append(attr)
134.
135.             print("system learned that " + self.commands[comm] + "
    " + str(self.get_attributes(returned_attr_num[comm])))
136.             if np.array_equal(desired_attr_num[comm],
    returned_attr_num[comm]):
137.                 commandslearned[comm] = True
138.             else:
139.                 print("This result is incorrect. The correct output
    should be: ")
140.                 print(self.commands[comm] + " " +
    str(self.get_attributes(desired_attr_num[comm])))
141.
142.
    desired_attr.append(self.get_attributes(desired_attr_num[comm]))
143.
    returned_attr.append(self.get_attributes(returned_attr_num[comm]))
144.
145.         return [attributeslearned, summedoutputs, [commandslearned,
    desired_attr, returned_attr]]
146.
147.
```

```
148.     def get_attributes(self, _attrindices):
149.         return_attributes = []
150.         for a in range(len(_attrindices)):
151.
   return_attributes.append(self.attributes.get(_attrindices[a]))
152.         return return_attributes
153.
154.
155. """
156. Preparing the Excel file.
157. Sheet 1 is overall success (per iteration, all attributes).
158. Sheet 2 is success per attribute.
159. Sheet 3 is success per command (e.g. "robin has")
160. """
161. wb = xlwt.Workbook()
162. sh1 = wb.add_sheet("successrate")
163. sh1.write(0,0,"id")
164. sh1.write(0,1,"cellsize")
165. sh1.write(0,2,"intsparsity")
166. sh1.write(0,3,"conn_strength")
167. sh1.write(0,4,"result")
168.
169. sh2 = wb.add_sheet("attributes")
170. sh2.write(0,0,"id")
171. sh2.write(0,1,"cellsize")
172. sh2.write(0,2,"intsparsity")
173. sh2.write(0,3,"conn_strength")
174. sh2.write(0,4,"attribute")
175. sh2.write(0,5,"convergence")
176. sh2.write(0,6,"c3_activation")
177. sh2.write(0,7,"summedoutputs")
178.
179. sh3 = wb.add_sheet("commands")
180. sh3.write(0,0,"id")
181. sh3.write(0,1,"cellsize")
182. sh3.write(0,2,"intsparsity")
183. sh3.write(0,3,"conn_strength")
184. sh3.write(0,4, "seed")
185. sh3.write(0,5,"command")
186. sh3.write(0,6,"expected_attributes")
187. sh3.write(0,7,"result_attributes")
188. sh3.write(0,8,"correct")
189.
190. print("Excel file created")
191.
192. """
193. Writes the results of the current iteration into an Excel file.
194. First sheet is overall success:
195.     Writes networkinfo in the correct columns.
196.     Checks convergence runs of all logic gates;
197.     all None = failed, no None = success, otherwise partial.
198. Second sheet is success per logic gate:
199.     Calculate the row to write on based on _row and the amount of
   logic gates solved.
200.     Write all data in the corresponding column.
201. """
202. def addresultstoexcel(_excelinfo, _networkinfo, _results, _row,
   _attributes, _commands):
203.     # _excelinfo    = [workbook, sheet1, sheet2, sheet3]
204.     # _networkinfo  = [id, cellsize, sparsity, conn_str, seeds]
```

```
205.     # _results      = [attributeslearned, summedoutputs,
   [commandslearned, expected_attr, returned_attr];
206.     #    attributeslearned is learningrun, commandslearned is
   true/false
207.     for i in range(len(_networkinfo)):
208.         _excelinfo[1].write(_row, i, str(_networkinfo[i]))
209.
210.     allsuccess = True
211.     allfailed = True
212.     for attribute in range(len(_results[0])):
213.         if _results[0][attribute] == None:
214.             allsuccess = False
215.         else:
216.             allfailed = False
217.     if allsuccess:
218.         _excelinfo[1].write(_row, len(_networkinfo), "success")
219.     elif allfailed:
220.         _excelinfo[1].write(_row, len(_networkinfo), "failure")
221.     else:
222.         _excelinfo[1].write(_row, len(_networkinfo), "partial")
223.
224.     # attribute sheet
225.     for attribute in range(len(_results[0])):
226.         subrow = ((_row-1) * len(_results[0])) + attribute + 1
227.         for i in range(len(_networkinfo)):
228.             _excelinfo[2].write(subrow, i, str(_networkinfo[i]))
229.         _excelinfo[2].write(subrow, len(_networkinfo),
   _attributes[attribute])
230.         _excelinfo[2].write(subrow, len(_networkinfo)+1,
   str(_results[0][attribute]))
231.         _excelinfo[2].write(subrow, len(_networkinfo)+2,
   str(_results[1][attribute]))
232.
233.     # commands sheet
234.     for command in range(len(_results[2][0])):
235.         subrow = ((_row-1) * len(_results[2][0])) + command + 1
236.         for i in range(len(_networkinfo)):
237.             _excelinfo[3].write(subrow, i, str(_networkinfo[i]))
238.         _excelinfo[3].write(subrow, len(_networkinfo),
   _commands[command])
239.         _excelinfo[3].write(subrow, len(_networkinfo)+1,
   str(_results[2][1][command]))
240.         _excelinfo[3].write(subrow, len(_networkinfo)+2,
   str(_results[2][2][command]))
241.         _excelinfo[3].write(subrow, len(_networkinfo)+3,
   str(_results[2][0][command]))
242.
243.
244. cellsize        = [10, 20, 50, 100]
245. intsparsity     = [.80, .85, .88, .92, .95, .98]
246. extsparsity     = [1]
247. conn_strength   = [0.2, 0.5, 0.8, 0.0] # 0.0 for random
248. # seeds           = [ [1,2,3,4],
249. #                     [5,6,7,8] ]
250.
251. resultscounter  = 145
252. for size in range(len(cellsize)):
253.     for intspars in range(len(intsparsity)):
254.         for strength in range(len(conn_strength)):
255.             # for seed in range(len(seeds)):
256.             print("Current iteration: " + str(resultscounter))
```

```
257.             print("Cellnetwork with nodes=" + str(cellsize[size]) +
   ", intspars=" + str(intsparsity[intspars]) +
258.                   ", conn_str=" + str(conn_strength[strength]))# +
   ", seeds=" + str(seeds[seed]))
259.             networkinfo = [resultscounter, cellsize[size],
   intsparsity[intspars], conn_strength[strength]]
260.
261.             cn = CellNetwork(cellsize[size], intsparsity[intspars],
   1, [0,0,0,0], conn_strength[strength])
262.             patterns = cn.get_patterns()
263.             results = cn.learn_pattern(patterns[0], patterns[1])
264.
265.             addresultstoexcel([wb, sh1, sh2, sh3], networkinfo,
   results, resultscounter,
266.                               cn.attributes, cn.commands)
267.             wb.save("results_v7.3b_variableseed.xls")
268.             print("Saved results to Excel")
269.             resultscounter += 1
```

Subcell

```
1. # -*- coding: utf-8 -*-
2. """
3. Created on Tue Apr 28 12:08:19 2020
4. Adaptation: 30-06-2020
5.
6. @author: Kevin Liu & Robbin Koopman
7. """
8.
9. import numpy as np
10.  import random
11.  import math
12.
13.  class Subcell():
14.
15.      def __init__(self, _cellnodes, _seedcombo, _intconnspars,
   _extconnspars, _connstr, _numattributes, _timesteps = 10, _inputnodes
   = 2):
16.          outputnodes = 1 # Amount of output nodes
17.          self.skip = -99
18.
19.          # array for storing all activation levels of all cellnodes
   at each timestep
20.          # for each inputpattern
21.          self.activation_cellnodes = np.zeros(_numattributes *
   _timesteps * _cellnodes).reshape(
22.                                               _numattributes,
   _timesteps, _cellnodes)
23.
24.          # Generating connection arrays
25.          self.inputconnections = self.networkgen(_inputnodes,
   _cellnodes, _seedcombo[0], _intconnspars, _connstr)
26.          self.nodeconnections = self.networkgen(_cellnodes,
   _cellnodes, _seedcombo[1], _intconnspars, _cellmidlayer=True)
27.          self.outputconnections = self.networkgen(_cellnodes,
   outputnodes, _seedcombo[2],  _intconnspars, _connstr)
28.          self.intercellconnections = self.networkgen(_cellnodes,
   _cellnodes, _seedcombo[3], _extconnspars, _connstr)
29.
30.          # Values for control;
```

```
31.           # controlactivation adds activation as additional input to
   all cellnodes
32.           # controlsignconnection inverts input activation
33.           self.controlactivation = 0.0
34.           self.controlsignconnection = 1.0
35.
36.       """
37.       Generates a network consisting of connections between
   _startnodes and _endnodes.
38.       Each _startnode can be connected to any number of _endnodes.
39.       Connectionstrength is the same for all connections; this can be
   changed later if needed.
40.       self.skip indicates no connection is made between _startnode[i]
   and _endnode[j].
41.       _cellmidlayer is only True when connecting cellnodes, as
   cellnodes should not be connected to themselves.
42.       In all other cases, it is false; e.g. inputnode 0 could be
   connected to cellnode 0.
43.
44.       Returns the generated network.
45.       """
46.       def networkgen(self, _startnodes, _endnodes, _seed, _threshold,
   _connstrength = 0, _cellmidlayer = False):
47.           dummyconnections = np.zeros(_startnodes *
   _endnodes).reshape(_startnodes, _endnodes)
48.           random.seed(_seed)
49.           for i in range(_startnodes):
50.               for j in range(_endnodes):
51.                   dummyconnections[i,j] = self.skip
52.                   if _cellmidlayer and j == i:
53.                           continue
54.                   if random.random() >= _threshold:
55.                       if not _connstrength == 0.0:
56.                           dummyconnections[i, j] = _connstrength
57.                       else:
58.                           dummyconnections[i, j] =
   round(random.random(),4)
59.           return dummyconnections
60.
61.       """
62.       Calculates the total activation onto cellnode q, given by:
63.       - the inputnode * controlsignconnection
64.       - activation of cellnodes k at timestep t connected to cellnode
   q * connectionstrength of k to q
65.       - activation of othercellnodes k at timestep t connected to
   cellnode q * connectionstrength of k to q
66.       Stores the squashed inputSum as activation of cellnode q in
   timestep t+1.
67.       For the last timestep, also apply controlactivation before
   squashing.
68.       Lastly, store the end activations into the activationperinput
   array (used by the getoutput function).
69.       """
70.       def spreadactivation(self, _timestep, _input, _inputnumber=0,
   _othercell=None, _controlinsquash=True):
71.           if _othercell != None:
72.               inputothercell =
   _othercell.activation_cellnodes[_inputnumber, _timestep]
73.           nextstep = _timestep + 1
74.           for q in range(len(self.activation_cellnodes[_inputnumber,
   _timestep])):
```

```
75.             inputsum = 0.0
76.             for k in range(len(_input)):
77.                 if self.inputconnections[k, q] != self.skip:
78.                     inputsum += _input[k] *
   self.inputconnections[k, q] * self.controlsignconnection
79.             for k in
   range(len(self.activation_cellnodes[_inputnumber, _timestep])):
80.                 if self.nodeconnections[k, q] != self.skip:
81.                     inputsum +=
   self.activation_cellnodes[_inputnumber, _timestep, k] *
   self.nodeconnections[k, q]
82.             if _othercell != None:
83.                 for k in range(len(inputothercell)):
84.                     if _othercell.intercellconnections[k, q] !=
   self.skip:
85.                         inputsum += inputothercell[k] *
   _othercell.intercellconnections[k, q]
86.             if _controlinsquash:
87.                 if nextstep ==
   (len(self.activation_cellnodes[_inputnumber])-1):
88.                     inputsum += self.controlactivation
89.
90.             self.activation_cellnodes[_inputnumber, nextstep, q] =
   round(math.tanh(inputsum),4)
91.
92.     """
93.     Calculates the output of the subcell;
94.     Takes the activations of each cellnode at the last timestep
   (stored in self.activationperinput),
95.     multiplies each activation by the connectionstrength of
   cellnode q on outputnode,
96.     and adds all the (activations*connectionstrengths) together.
97.
98.     Returns the sum of all (activations*connectionstrengths)
99.     """
100.    def getoutput(self, _inputnumber=0):
101.        outputsum = 0.0
102.        for k in range(len(self.activation_cellnodes[_inputnumber,
   -1])):
103.            if self.outputconnections[k, 0] != self.skip:
104.                outputsum +=
   self.activation_cellnodes[_inputnumber, -1, k] *
   self.outputconnections[k, 0]
105.        return round(outputsum, 4)
106.
107.    """
108.    Solves the intermediate logic gates.
109.    Requires the inputpattern, pattern to be solved and the
   learningrate.
110.    Adjusts the input signconnection if input (0,0) expects an
   output > 0.
111.    Checks for error in the outputpattern and adjusts
   controlactication accordingly.
112.    Returns True when the subcell converged, the newly adjusted
   controlactication, and the outputs of the subcell.
113.    """
114.    def learn_subpattern(self, _input, _desiredoutput,
   _learningrate):
115.        learned = True
116.        errvalue = 0
117.        summedoutputs = np.zeros(len(_desiredoutput))
```

```
118.         outputpattern = np.zeros(len(_desiredoutput))
119.
120.         for i in range(len(_input)):
121.             # Check if input = [0,0] desires output > 0. If yes,
    case must be NAND/NOR, thus reversesignactivation is called.
122.             if (_input[i] == np.array([0,0])).all():
123.                 if _desiredoutput[i] > 0:
124.                     self.reversecontrolsignconnection()
125.
126.             summedoutputs[i] = self.getoutput(i)
127.             if summedoutputs[i] > 0:
128.                 outputpattern[i] = 1
129.             else:
130.                 outputpattern[i] = 0
131.
132.         # Check for error in outputpattern
133.         for i in range(len(_desiredoutput)):
134.             error = _desiredoutput[i] - outputpattern[i]
135.             if error != 0:
136.                 learned = False
137.                 errvalue = error
138.
139.         # Apply learningrate to cellnodes
140.         if errvalue != 0:
141.             c3c4_activation = _learningrate * errvalue
142.             self.adjustcontrolactivation(c3c4_activation)
143.
144.         return [learned, self.controlactivation, summedoutputs]
145.
146.     """
147.     These functions represent the control nodes of the subcell;
148.     adjustcontrolactivation adds (error margin * learningrate)
    provided by the learning function to self.controlactivation.
149.     applycontrolactivation is only used for applying control after
    squashing. Should be called after spreading activation.
150.     reversecontrolsignconnection sets self.controlsignconnection to
    -1, thus inverting the input given to the cell only once.
151.     """
152.     def adjustcontrolactivation(self, _adjustment):
153.         self.controlactivation = round(self.controlactivation +
    _adjustment,4)
154.     def applycontrolactivation(self, _inputnumber):
155.         for c in range(len(self.activationperinput[_inputnumber])):
156.             self.activation_cellnodes[_inputnumber, -1, c] +=
    self.controlactivation
157.     def reversecontrolsignconnection(self):
158.         self.controlsignconnection = -1
```