CONSTRAINT PROGRAMMING TECHNIQUE FOR THE AUTOMATED COMBINATION OF NETWORKS OF FUNCTIONAL GROUPINGS OF SPIKING NEURONS

by

Adam Bennett

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Carleton University
Ottawa, Ontario
April 2018

Table of Contents

List of Tables

List of Figures

Abstract

Spiking neural networks have been shown to be more powerful computationally than traditional rate-coded neural networks, but they can be very difficult to train to perform specific tasks. Existing training techniques tend to be monolithic in nature and scale poorly to larger networks.

This thesis aims to address this issue by providing an automated technique for combining certain types of functional groupings of spiking neurons into composite functional networks. The technique takes the form of an algorithm which uses constraint programming to ensure that four axioms hold true in the network; the axioms having been designed to ensure that signals arrive simultaneously to component groupings. A number of experiments were conducted in which the algorithm was used to combine component groupings into more complex composite networks; these experiments show the practical utility of the algorithm and reinforce by demonstration the correctness of the axioms. Networks designed for the experiments include memory registers which store binary values, binary counters, and a robot-control network for navigating a generalized variant of the maze described by the T-Maze problem.

Acknowledgements

The author wishes to thank the following individuals who made this thesis possible:

Dr. Anthony White for his guidance, patience, and the occasional cup of coffee. His experience and wisdom were invaluable during the course of this project, and I could always count on him for support, direction and motivation when I was unsure of how to proceed.

My parents, David and Diane Bennett, for their tremendous support and understanding. They were always willing to listen to me talk about the latest problem in my research, and never doubted that I could solve it.

The members of Carleton's Complex Adaptive Systems group, for always being quick to offer feedback and encouragement.

Chapter 1

Introduction

The organic brain is composed of a large number of neurons interconnected by synapses. Each individual neuron (see Figure 1.1 for a diagram of a biological neuron) is responsible for performing some small computation. Together, the computations performed by neurons and the connections between them are thought to form the basis of cognition. Artificial neural networks (ANN) are an abstraction of the organic brain. Using a network of artificial neurons and synapses (which mimic their biological counterparts), ANNs have demonstrated considerable utility in solving real-world problems (i.e., classification problems, speech and vision processing, etc.). Typical ANN architectures have neurons arranged into distinct layers, with synapses connecting neurons in one layer to those in a subsequent layer and not connecting any two neurons which are in the same layer. Early architectures had only a single layer of neurons, while later architectures often had two or three. Deep learning ANN architectures may have many layers, those layers at the beginning of the network are used to form general, high level representations and subsequent layers form more specific representations.

Spiking neural networks (SNN) (see Chapter 2, Section 2.3 for more details) are a type of ANN which have been shown to be more powerful computationally than traditional neural network models (Maass, 1997). However, there still is not an all-purpose learning technique usable for training artificial SNNs comparable to the well known and widely used back-propagation algorithm (see Chapter 2, Section 2.2.1) used for training other types of ANN.

Chapter 3, Section 3.2.1 provides an overview of some of the various techniques for training SNNs; it also provides some examples of why SNNs tend to be more difficult to train than regular ANNs. While these techniques allow SNNs to be trained to perform complex tasks, they tend to train networks monolithically and are not easily

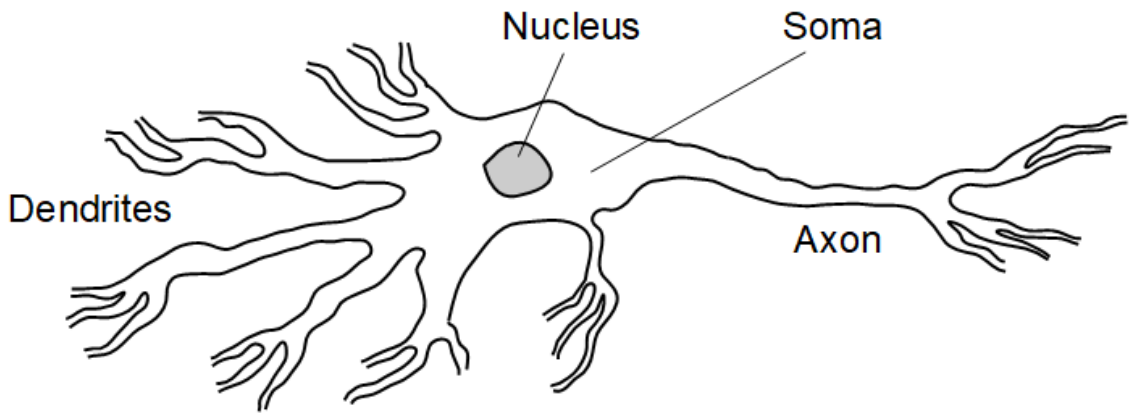


Figure 1.1: Diagram of a biological neuron. The axon is connected to the dendrites of other neurons by synapses (not shown)

scaled to large problems. Following from the research done by Jeanson in (Jeanson, 2013), the purpose of the research carried out in this thesis was to attempt to provide a method for combining multiple SNNs into a larger network that uses the sub-networks (also referred to as neuronal groupings) to perform subtasks of some overall problem. Unlike deep learning neural networks, the network is arranged not as layers representing different levels of abstraction but rather as a network of neuronal groupings which perform specific tasks reminiscent of Brooks' subsumption architecture (Brooks, 1990). The groupings would each be pre-trained to perform their given subtask before being combined. This would allow groupings to be used in a hierarchy to solve larger problems more easily if the problem could be decomposed into smaller sub-problems. Jeanson identified two categories of behaviour that a neuronal grouping may exhibit: reactive behaviour where the grouping produces one or more output pulses in accordance with the provided input pulses, and adaptive behaviour, where cycles of neurons can maintain a self-sustaining series of pulses to be used for dynamic memory (Jeanson, 2013). The method for combining neuronal groupings should be capable of combining groupings exhibiting either type of behaviour (including combining reactive groupings with adaptive memory groupings).

The main idea behind this technique for combining neuronal groupings was that if spike pulses being inserted into the inputs of a particular neuronal grouping arrive simultaneously, then the grouping would behave as it normally would outside of the

composite network. The challenge was then to ensure that spike pulses being output from other groupings at different times would arrive at their destination grouping at the same time. With this idea in mind, constraint programming (see Chapter 2, Section 2.4 for more details) was selected as a promising tool for ensuring that these rules were followed.

1.1 Objectives

The primary objective for the research reported in this thesis was to develop an automated technique for combining multiple functional groupings of spiking neurons into a larger network. In addition, the automated technique should work on the widest range of neuronal groupings as possible (for example, it should not be limited to groupings with a feed-forward topology), and it should be hierarchical (the network resulting from the combination should be usable as a grouping in an even larger network).

Since constraint programming was selected as a method to help develop the automated technique described in the primary objective, a secondary objective became necessary. The secondary objective was to form a formulaic description of the behaviour of spike pulses in a directed, cyclic network of spiking neurons. This description would be needed to help ensure that pulses arrive at the inputs to the functional groupings being combined with the correct timing needed for them to properly carry out their functions.

The above objectives only account for the design of SNNs in terms of setting delay lengths (see Chapter 2, Section 2.3 for more details). They assume that the user of the automated technique has already determined how the functional neuronal groupings should connect in terms of which outputs of a given grouping are connected to which inputs of another. This leaves the non-trivial problem of assigning appropriate delay lengths to the inter-grouping synapses to be solved by the automated technique. A tertiary objective was to find a technique to assign the start and end points for inter-grouping synapses in such a way that the resulting network (after using the technique designed for the primary objective to set the delay lengths for the inter-grouping synapses) can perform some specified function by using a combination of sub-networks.

1.2 Problem Statement

Can an automated technique for setting delay lengths of inter-grouping synapses in a network of functional groupings of spiking neurons be developed using constraint programming such that the network works as a composite of the individual groupings?

This problem can be divided into subtasks, each of which builds on the previous subtask:

- Ensuring that all given signals being inserted into a grouping arrive simultaneously for any grouping in a network of groupings and inter-grouping synapses where there are no cycles of groupings and groupings do not contain cycles of neurons (Chapter 4, Section 4.5, Axiom 1)
- As the first subtask, except the network may contain cycles of groupings (Chapter 4, Section 4.5, Axiom 2)
- As the second subtask, except that groupings may contain cycles of neurons (Chapter 4, Section 4.5, Axiom 3)
- As the third subtask, except that the resulting composite network must itself be usable as a grouping that can be combined into more complex networks (Chapter 4, Section 4.5, Axiom 4)

1.3 Contributions

This thesis makes the following contributions:

- Definition of Spiking Neuronal Components (referred to as components in this thesis), a type of functional grouping of spiking neurons with properties which make them relatively simple to combine into larger networks (Chapter 4, Section 4.2)
- Generalization of synfire chains (see Chapter 3, Section 3.2.4) to apply to chains of components with inter-component and intra-component recurrence (Chapter 4 Section 4.5)

- Algorithm for automatically setting delay lengths of inter-component synapses such that spike pulses arrive to a component's input neurons simultaneously. (Chapter 4, Section 4.5.1). Algorithm correctness is proven (Chapter 4, Section 4.5) and utility of the algorithm is demonstrated in solving a number of design problems (Chapter 5)

1.4 Organization

This thesis is organized as follows: Chapter 2 provides an overview of artificial neural networks and in particular spiking neural networks. It also covers the topic of constraint programming. Those knowledgeable about neural networks and constraint programming may bypass this chapter. Chapter 3 examines work related to this thesis, including modular neural networks, SNNs and their behaviour, topology, training, and applications, and the field of neuronal assembly computing. It explores how these topics relate to the research carried out in this thesis, and how the research in this thesis can benefit from these techniques and vice-versa. Chapter 4 describes the technique for combining SNNs which was developed as part of the research undertaken in this thesis and provides both a rationale explaining why the technique may be expected to work and algorithm specifications designed to implement the technique. Experiments were conducted to test the technique, these are detailed in Chapter 5 along with their results and observations regarding their performance. Finally, Chapter 6 summarizes the research carried out in this thesis, its applications, and future work that could build off of it.

Chapter 2

Background

2.1 Introduction

This chapter provides an overview of topics which are needed to understand the rest of the material in this thesis. Section 2.2 gives a high-level description of artificial neural networks. Section 2.3 describes SNNs, their properties, and several models used to simulate spiking neurons. Section 2.4 gives an introduction to constraint programming and how it works. Those who are knowledgable about these topics may bypass this chapter without loss of understanding the research described in subsequent chapters.

2.2 Artificial Neural Networks

ANNs use a particular representation to create intelligent systems. The algorithms used in ANN learning are studied in a branch of Machine Learning, which is considered part of the study of Artificial Intelligence. ANNs are loosely based upon a biological metaphor: the neuron (see Figure 1.1 in Chapter 1). Neuron models are very abstract. In its most general form, an ANN is a distributed network of simulated neurons which each perform some small computational task in a larger computation performed by the network as a whole. Artificial neurons (referred to as "neurons" in this thesis) are connected to each other by artificial synapses (referred to as "synapses" in this thesis) which are typically unidirectional and often have an associated weight value representing the strength of the connection.

One of the earliest and most well known neuronal models used in ANNs is the perceptron (Minsky et al., 2017) (see Figure 2.1). A perceptron models a neuron which has the following attributes:

- input synapses, each of which has a weight value and conveys a signal value either from another perceptron or an input. Positive weight values indicate

excitatory connections, while negative weight values indicate inhibitory connections. The strength of the connection corresponds to the magnitude of the weight

- output synapses, each of which has a weight and conveys the perceptron's output value to other perceptrons or outputs
- an activation function which specifies the perceptron's output value according to the weighted sum of input values. Earlier models used a simple stepwise function, while more complex models use sigmoidal or other non-linear functions. Non-linear functions, including sigmoidal functions, hyperbolic tangents, and softmax functions, are functions whose outputs are not directly proportional to their inputs
- a bias value, used to offset the weighted sum of inputs (often implemented as a weight along a synapse which originates at a special neuron which always outputs a value of 1 and leads to the perceptron whose weighted sum it offsets)

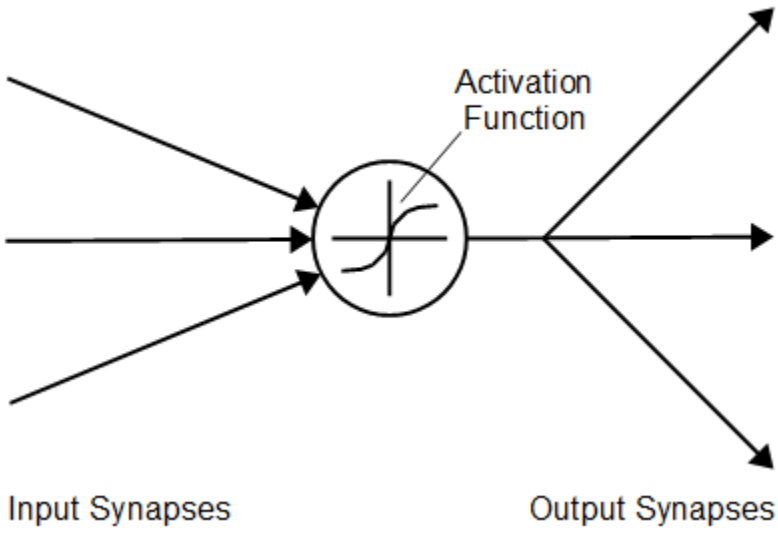


Figure 2.1: Perceptron neuronal model

A single perceptron can be trained to differentiate items of linearly separable classes by finding appropriate values for the synapse weights. However, a single

perceptron cannot be trained to classify items from classes which are not linearly separable. By using an ensemble of perceptrons arranged appropriately, it is possible to classify data which is not linearly separable.

Often, networks of perceptrons (or other neuronal models) are arranged into three groupings: the input layer, hidden layer(s), and output layer (see Figure 2.2). The input layer takes input values and passes them to the hidden layer. Inputs often take the form of vectors, where each element is inserted into a different input in the input layer. The hidden layer processes the data and presents the result to the output layer. Often, output values of output neurons are used as elements in a vector. The network is usually feed-forward, meaning that signals always travel from one layer to a subsequent layer, and never to the same layer or a previous layer.

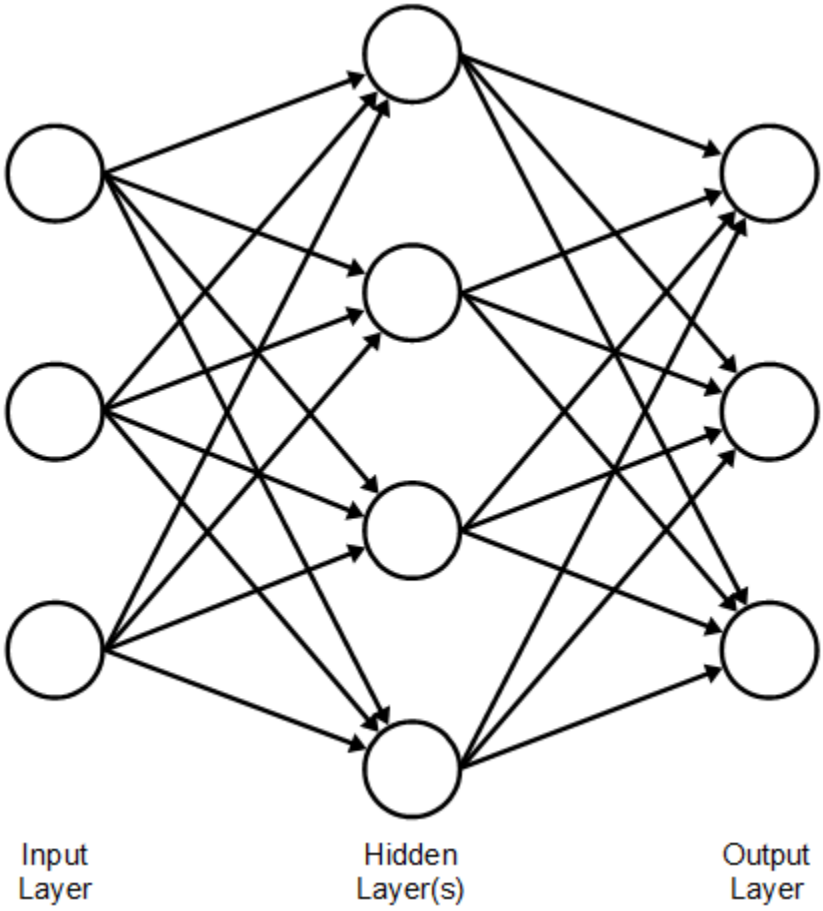


Figure 2.2: Example of a neural network with three layers

More complex ANN models exist, including those which have multiple hidden

layers, different activation functions, and networks which are not feed-forward. For example, deep learning neural networks are ANNs which have many hidden layers, and each layer represents a feature at a different level of abstraction. Another example of an ANN model is time delay neural networks (TDNN). TDNNs are useful when inputs to the network represent a time-series, such as in speech recognition tasks. The TDNN works by presenting inputs to the network corresponding to various time windows; this allows the network to consider input features independent of the time when they occur.

Traditionally, the output of a neuron in ANNs is used to model the rate at which the neuron fires (for example, a neuron which has an output of 0.2 for a given simulation might represent a neuron which is firing with a frequency of 2 pulses per millisecond). This is based on biological neurons, which have been observed firing at higher rates when they receive higher levels of input stimulation. This type of ANN model is referred to as a rate-coded network. In rate-coded networks, signals propagate instantaneously from one neuron to another; this is an abstraction as in biological neural networks there is a measurable delay in the length of time it takes a signal to propagate along a synapse.

2.2.1 Back Propagation

ANNs can solve complex problems, but only if the synapses have particular weights which depend on the problem to be solved. Finding the appropriate set of synapse weights for an ANN can be accomplished in many ways, but one of the most well known techniques is the back propagation algorithm.

In the back propagation algorithm (Rumelhart et al., 1986), synapse weights are traditionally assigned random initial weights (although there are other strategies for initializing weights (Bengio and Glorot, 2010)). A number of training data tuples consisting of input values and expected output values are fed to the algorithm one by one. The input values are inserted into the ANN's input nodes, and the resulting output values are compared to the expected output values. An error value is determined by comparing the expected output values provided as part of the training data to the actual output values computed by the ANN. An example of a typical error measure is provided below:

$$E = \frac{1}{2} \sum_{c} \sum_{o} (a_{o,c} - d_{o,c})^2, \tag{2.1}$$

where E is the total error of the network for all training tuples, c is an index over each training tuple, o is an index over all neurons in the output layer, $a_{i,c}$ is the output value for the i^{th} neuron after running the c^{th} training tuple forward through the network, and $d_{o,c}$ is the expected output of the o^{th} output neuron for the c^{th} training tuple (Rumelhart et al., 1986).

After calculating the error for a training data tuple, the weights of the synapses in the network are updated one layer at a time in an attempt to reduce the error value for that tuple. First, the weights of all synapses leading from the last layer of hidden neurons to output neurons are updated. Next, the weights for synapses leading from the second last hidden layer to the last hidden layer are updated. This continues backwards through the network until all layers have been updated. Updating weights is done using a process called gradient descent. Gradient descent works by using partial derivatives to determine how changes to a given synapse's weight will affect the error value. Weights are then changed in a way that minimizes the error. The equation to update a synapse's weight is provided below:

$$\Delta w_{ij} = -\epsilon \frac{\delta E}{\delta w_{ij}}, \tag{2.2}$$

where w_{ij} is the weight of the synapse originating at neuron i and leading to neuron j, ϵ is the learning rate (a constant used to modify the magnitude of changes to the weights), and $\frac{\delta E}{\delta w_{ij}}$ is the partial derivative of E with respect to w_{ij}.

Once the weights of all synapses have been updated, the next training tuple is run through the network and the weights are updated again. This continues until the network has been satisfactorily trained to solve the given problem. Note that there are variations on back propagation. For example, another equation for updating weights is provided in (Rumelhart et al., 1986) which computes the accumulated error for all training tuples before updating any weights.

2.3 Spiking Neural Networks

A type of ANN which does not use activation levels to represent firing rates, called spiking neural networks (also referred to as precise spike timing ANNs, delay-coincidence detection neural networks, third generation neural networks, and temporally coded neural networks), associates a delay length to each synapse which represents the length of time it takes for a signal to travel from one neuron to another. Although there are numerous neuronal models for SNNs, one thing they all have in common is that they all rely on the timing of signals (also referred to as "pulses") arriving at neurons for computation. In general, a neuron will fire (send pulses to all neurons it outputs to) if and only if it receives a sufficient number of input pulses (potentially modified by synapse weight) either simultaneously or over a short time period to reach its activation threshold.

SNNs have some advantages over rate coded neural networks. SNNs have been proven to be more computationally powerful than rate coded networks (Maass, 1997). They also provide a solution to the binding problem (Fujiia et al., 1996). The binding problem questions how a neural network can represent objects with multiple properties. For example, if an object has the properties "shape" and "colour", how would a neural network represent objects with permutations of the possible properties? In rate coded networks, a separate representation is needed for each possible permutation, which is highly impractical. SNNs, however, can have a representation of each possible value of each attribute and can represent an object with a mix of attributes by having the individual representations of the attribute values firing at the same frequency. This means that it would be possible that the object would have an associated frequency and it can be bound to the attributes oscillating on that frequency.

In practice, SNNs can be used for a wide variety of applications. Tasks which SNNs can be applied to include, but are not limited to, classification, pattern recognition and learning, and reactive control systems which make use of dynamic memory.

As mentioned above, there are many different types of SNN. In this thesis, a modified (see Section 4.3 in Chapter 4) version of the Leaky Integrate-and-Fire (LIF) neuronal is used. LIF neurons are defined as follows (Jeanson, 2013):

$$a_i(t_n) = \theta[\sum_j w_{ij} a_j(t_n - d_{ij})] \; if \; t_n - (R + t_{s_i}) > 0$$
$$but \; a_i(t_n) = a_i(t_{n-1}) \; if \; f(a_i(t_n - 1)) > 0, \tag{2.3}$$

where $a_i(t_n)$ is the activation state of neuron i at time step n, j is an index over all neurons with outgoing synapses leading to neuron i, w_{ij} is the weight of the synapse from neuron j to neuron i, and d_{ij} is the delay time of the synapse from neuron j to neuron i. θ is the activation function for neuron i, which returns 1 if the parameter is greater than or equal to neuron i's activation threshold and 0 otherwise. R is the refractory time, which specifies how long a neuron must remain idle after it finishes firing before it can fire again. $f(a_i)$ is the leak state of neuron a_i; which may be 1 or 0. After firing, a neuron enters a leak state ($f(a_i) = 1$) where it continues to output its last signal until it leaves its leak state ($f(a_i) = 0$) after a set amount of time.

LIF neurons are useful because they are not as computationally expensive to simulate as more biologically realistic neuronal models. They are especially useful in this thesis since their simple nature allows for simpler formulation of constrained equations in Chapter 4.

2.4 Constraint Programming

Constraint programming is a technique where a problem is solved by presenting a series of constraints that must hold true for potential solutions to the problem. A program which searches for solutions to a problem specified using constraint programming is called a constraint solver. In the research presented in this thesis, the Choco constraint solver (Prud'homme et al., 2017) was used.

In constraint programming, a set of constraints must be provided to the constraint solver. The constraints may take the form of equations or inequalities with variables, and must hold true for all possible solutions to the problem in question (or at least all of the solutions of interest to the user). In addition, the user may provide an objective: guidelines to the constraint solver for prioritizing certain solutions over other solutions. For example, in this thesis the constraint solver was instructed to find the solution which had the lowest possible sum of variable values while still meeting the constraints.

The Choco solver works by alternating between constraint propagation and large neighbourhood search (Prud'homme et al., 2017). Constraint propagation works by treating some variables as having independent domains and others as having domains based on the values of other variables, as determined by the set of constraints. For example, if a problem had two variables, x and y (with domain equal to 0→100), and the constraint $x \leq y$, then the constraint propagation algorithm may treat y as having an independent domain and x as having the dependent domain $0 \rightarrow y$. In constraint propagation, whenever a value is selected for a variable, other variables with dependent domains based on that variable have their domains recalculated. Large neighbourhood search is a local search technique which works by relaxing one or more variables in a solution to their domain while leaving others set. It then chooses new values for the relaxed variable(s) in order to arrive at an adjacent solution in the solution space. Although the Choco solver allows certain elements of its solving algorithm to be changed or specified by the user, the default settings were used for this thesis.

The following is an example of a problem may be solved using constraint programming. Consider the problem of paying for a purchase with a combination of coins; the goal is to pay using the minimum total number of coins. The total cost of the purchase is \$1.28. Three quarters, five dimes, seven nickels, and thirty pennies are available for payment. This problem can be expressed using constraint programming using a combination of variables, constraints, and an objective. The variables consist of variables assigned to each type of coin (pennies, nickels, etc.) whose values represent the number of the type of coin in question being spent (see Table 2.1). The set of constraints (see Table 2.2) consists of one constraint stating that the total value of coins spent must equal \$1.28 and a constraint for each coin type stating that one cannot spend more of that coin type than are available. Finally, an objective is needed to drive the constraint solver to an optimal solution. Although the coins available provide a multitude of possible solutions which will all satisfactorily pay the \$1.28 cost, it was stated in the problem description that a minimum number of coins should be used. Therefore, the objective for the problem is $min(V_P + V_N + V_D + V_Q)$. Using the variables, constraints, and objective gives the following optimal solution: $V_P = 3, V_N = 0, V_D = 5, V_Q = 3$.

Variable	Description
V_P	Number of pennies to spend
V_N	Number of nickels to spend
V_D	Number of dimes to spend
V_Q	Number of quarters to spend

Table 2.1: Example variables for the coin problem described in Section 2.4

Constrained Equation/Inequality	Rationale
$V_P \leq 30$	Only 30 pennies are available to spend
$V_N \leq 7$	Only 7 nickels are available to spend
$V_D \leq 5$	Only 5 dimes are available to spend
$V_Q \leq 3$	Only 3 quarters are available to spend
$V_P + 5 \times V_N + 10 \times V_D + 25 \times V_Q = 128$	The total value of coins spent must equal \$1.28

Table 2.2: Example constraints for the coin problem described in Section 2.4

Chapter 3

Related Work

This chapter examines research related to the design of spiking neural network topologies, with a focus on techniques which allow for SNNs to be built in a modular way. The first section discusses general techniques for combining artificial neural networks, including those which are not specific to spiking networks. The second section addresses works relating to established spiking neural network topologies which may contain techniques useful for designing modular networks and/or techniques which may benefit from the technique described in Chapter 4.

3.1 Modular Neural Networks

3.1.1 Ensemble Networks

One common approach to combining neural networks (or other machine learning techniques) is to use an ensemble network (Valentini and Masulli, 2002). In this thesis, only ensemble networks of neural networks are discussed. An ensemble network is a collection of independent neural networks which are each trained and run simultaneously along with some aggregation mechanism for selecting which network's output to use in a given context (Hansen and Salamon, 1990). The aggregation mechanism can come in many varieties. It may be as simple as taking the average or median of the output values provided by all constituent networks, or as complex as having a secondary machine learning technique which is trained to select an appropriate member of the ensemble depending on the circumstances. In addition, it is sometimes best to use only a subset of the networks in the ensemble determined based on input as opposed to using all trained networks (Zhou et al., 2002).

Traditionally, ensemble methods in neural networks have been used to improve classification accuracy; having many networks reduces the probability that all networks will find a solution in a local minima (Hansen and Salamon, 1990). In some

cases, an approach similar to ensembles is used to combine multiple neural networks which have been trained on a sub-problem found by decomposing the original problem; this approach is known as a mixture of experts.

Ensemble methods can be combined to form hierarchies. Sometimes, this involves having an ensemble learner which aggregates over a number of sub-ensembles (Guo, X and Gao, X, 2008). Other times the ensemble combines the results of learning techniques which operate at different levels of abstraction (Su et al., 2009).

3.1.2 Mixture of Experts

Many problems can be solved more easily by using the divide and conquer approach. Some problems can be decomposed automatically, such as with classification problems with more than two classes (Lu and Ito, 1999).

In some cases, a problem may have an obvious or intuitive breakdown. For example, a mixture of experts approach was applied to the problem of self-driving cars (Pomerleau et al., 1991). In that work, a number of neural networks were trained to operate a vehicle, with each network specializing in operating the vehicle under different conditions (i.e., dirt road, two lane highway, etc.) or using different inputs (i.e., visible spectrum, laser range-finder, laser-reflectance sensor). At any given moment, an appropriate network is selected to control the vehicle according to current conditions by an arbitrator. The arbitrator is not itself a neural network, but a rule-based system which uses symbolic logic to assign the active expert. Pomerleau et al. make note of the difficulties inherent in training neural networks to perform tasks based on symbolic reasoning using a purely connectionist paradigm.

A significant drawback of ensemble techniques and mixtures of experts is that the results of individual expert networks must be interpreted in some way (and potentially converted/encoded into some other representation to be fed into other networks). For example, a feed-forward network's output vector entries might be interpreted as bits in a binary number. This can be a problem for neural networks which represent data in complex or non-obvious ways. This is especially problematic in spiking neural networks, since data is often represented using time (for example, a single output neuron could represent a binary number by interpreting its spiking pattern over a period of time) and can be challenging to interpret or encode.

3.1.3 Networks of Networks

A network of networks is a neural network architecture where a network contains nested neuronal groupings (and these nested groupings may contain their own sub-groupings, and so on) and any connections between any groupings are at the neuronal level (Sutton et al., 1988)(Ling et al., 1997). In other words, a neuron in a grouping may connect to other neurons in its own grouping, or to specific neurons in other groupings; the "output" of a sub-network is not interpreted in any specific way.

3.1.4 Connectionist Glue

Connectionist glue (Waibel, 1989) is a process used to connect multiple pre-trained neural networks together. To apply the technique, the pre-trained networks are frozen, meaning their topology and parameters are fixed. Next, a "glue" layer of randomly initialized neurons and connections are inserted in between and connecting to the inputs and outputs of the initial networks. Finally, the network is re-trained (keep in mind that the original networks are not changed in any way) so that the glue layer can learn the appropriate weights to connect the initial networks. This technique proved to be as effective as training a neural network on the more complex, combined, problem while being much faster (due to the subtasks having already been learned individually).

The work done by Waibel in (Waibel, 1989) specifically used time-delay neural networks (not to be confused with spiking neural networks). A time delay network is a feed forward neural network used to find patterns in data which has a time element. Data from different windows in time are presented simultaneously to TDNNs using different weights to signify the time of the window. Although it was used by Waibel to combine TDNNs, it is reasonable to hypothesize that a similar technique may work on other neural network architectures. This thesis specifically makes use of the idea of freezing the initial networks after training them on a sub-problem.

3.2 Spiking Neural Networks

3.2.1 Learning Techniques

A brief overview of some common learning techniques for spiking neural networks is given here.

Back Propagation

Unlike traditional rate-based neural networks, which typically use continuous values to represent data and do not incorporate a time element in computation, spiking neural networks rely on the timing of discrete signals which take a specific amount of time to travel through the network for computation. Neurons in SNNs have activation functions which are not differentiable; this makes applying some common learning techniques for rate-based networks (such as back propagation) difficult (Lee et al., 2016).

Despite this difficulty, a number of works have been presented which modify the back propagation algorithm to work in SNNs (Bohte et al., 2002) (Lee et al., 2016). This is typically accomplished by using a differentiable function that relates in some way to the behaviours of the spiking neurons. For example, in (Bohte et al., 2002), the non-differentiable spike trains of spiking neurons are approximated as distributions which are differentiable. Despite some success, back propagation encounters a number of issues in SNNs which do not occur in rate-based networks, such as increased error rates appearing during training (Shrestha and Song, 2017).

Another limitation of back propagation in SNNs is that most implementations of it do not account for networks with recurrent structures in their topology. Back propagation can be applied to recurrent networks by "unfolding" recurrences in the network; essentially treating the recurrent network as being a number of sequential feed-forward networks (Tino and Mills, 2006). While this technique shows some promise, the complexity of the training algorithm may make it difficult to scale and impractical in larger networks.

Despite the difficulties mentioned above, back propagation can (with modification) still be used as a reliable supervised learning algorithm to train feed-forward SNNs (Bohte et al., 2002).

Evolutionary Learning Techniques

Evolutionary algorithms provide another option for supervised training of SNNs. A differential evolution algorithm has been shown to work well for classification tasks (Pavlidis et al., 2005); however this work only considers feed-forward SNNs. Evolutionary algorithms have also been used to train the output layers of reservoir networks (Kasabov et al., 2014). In this case, the evolutionary algorithm is not used to train a recurrent network, but rather to interpret the output of a recurrent network that can compute complex functions involving memory (see Section 3.2.2 for more details on reservoir computing).

SNNs are useful for both adaptive and reactive tasks, such as mobile robot control; evolutionary algorithms are capable of allowing SNNs to develop complex behaviours (Jeanson, 2013) (Eskandari et al., 2016b) (Kubota and Sasaki, 2005).

ReSuMe

Another supervised learning technique, ReSuMe (Remote Supervised Method) (Ponulak, 2005), is able to produce specific output spike-trains when given corresponding input spike-trains as well as perform more traditional classification tasks. ReSuMe uses a constraint system to apply changes to synapse weights in relation to a provided "teaching" signal. ReSuMe is interesting because it was developed specifically to train SNNs, rather than being a modified rate-coding technique (like back propagation) or general-purpose technique (like evolutionary algorithms. An interesting variant of ReSuMe, DL-ReSuMe (Taherkhani et al., 2015), was introduced as well. DL-ReSuMe is unique among many learning techniques for SNNs in that it directly modifies the delay length of synapses in addition to weights; most training techniques tend to just modify weights.

Spike-Time Dependent Plasticity

Spike-time dependant plasticity (STDP) is an approach which has proven useful for training spiking neural networks (Feldman, 2012). STDP, a form of Hebbian learning (Hebb, 1949), works by strengthening (i.e., increasing the weight of) the synapses which contribute to the firing of a neuron. Although typically used in unsupervised

learning (Liu and Yue, 2017) (Diehl and Cook, 2015), STDP has been adapted for use in supervised learning scenarios as well (Hu et al., 2017) (Shrestha et al., 2017).

3.2.2 Reservoir Computing

Artificial neural networks, including SNNs, often operate as black boxes. There is an approach for recurrent neural networks which embraces this property; it is known as reservoir computing. A reservoir neural network (Lukoševičius and Jaeger, 2009) is a neural network containing recurrence, and whose outputs have been trained to identify some specific phenomena. The internal structure of the network is generated randomly, without any specific goal in mind; the recurrent nature of the "reservoir" is sufficient to obtain a variety of nonlinear computations. SNNs are often used as reservoirs for reservoir computing (Burgsteiner et al., 2007) (Verstraeten et al., 2005) (Schliebs et al., 2011); these reservoir networks would fall under the liquid state machine (LSM) category of reservoir computing approach (see below). The output is the only part of the network which is trained. This section describes the two main types of reservoir computing: liquid state machines and echo state networks.

Although it shows considerable promise for solving complex problems, the black-box nature of reservoir computation does result in issues relating to scalability. As the dimension of input and output vectors for a problem grow, the difficulty of training a reservoir network to properly encode and decode information to and from the reservoir increases (Luitel and Venayagamoorthy, 2014). The work done by Luitel and Venayagamoorthy shows that using a modular approach wherein a number of recurrent neural networks are sparsely connected to each other, where each module acts as a reservoir, can be used to improve scalability of reservoir computation methods.

Liquid State Machines

LSMs use biologically realistic neuronal models and were originally used for studying the biological processes of the brain (Lukoševičius and Jaeger, 2009). In exchange for a more biologically plausible model, they require more processing power to compute.

Interestingly, the performance of LSMs can be improved by applying unsupervised STDP learning to the reservoirs (Wang and Li, 2016).

Echo State Networks

Echo state networks are simpler and less costly to run than LSMs, and operate using a combination of vectors and nonlinear activation functions common among ANN models.

3.2.3 Dynamical Cell Assemblies

A key idea in the field of spiking neural networks is the dynamical cell assembly hypothesis (Fujiia et al., 1996). This hypothesis proposes that neurons may belong to one or more assemblies: groupings of neurons whose firing behaviours are linked together. Neurons in a dynamic assembly do not necessarily fire synchronously, nor are they all necessarily directly connected to one another by synapses. Rather, they are functionally connected: when one neuron fires, it may indirectly cause another neuron in the same assembly to fire later. This can continue to propagate throughout the assembly.

3.2.4 Synfire Chains

Synfire chains (Hayon et al., 2005) are feedforward networks of groupings of neurons where all neurons in a grouping fire simultaneously, sending signals to the next grouping in the chain which causes neurons in that grouping to also fire simultaneously; this behaviour continues from one grouping to the next until the end of the chain is reached (see Figure 3.1). One of the main advantages of a chain of groupings of neurons compared to a chain of single neurons is the robustness to noise provided by large numbers of cooperating neurons (Griffith, 1963).

Synfire chains have been studied in attempts to understand the behaviour and dynamics of networks in which they are found. This can be difficult, because finding synfire chains is a challenge due to the size and complexity of the networks they are used to examine. To this end, a large amount of research (Gerstein et al., 2012) has been conducted with the intention of finding automatic techniques for identifying synfire chains in large networks.

Synfire chains have also been used to create complex, recurrent spiking networks by combining a number of them together at random (Trengove et al., 2016). The

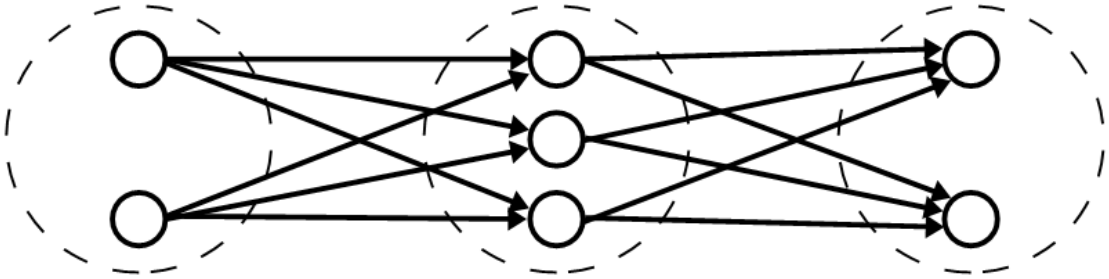


Figure 3.1: Diagram of a synfire chain. Neurons in the leftmost grouping are connected to neurons in the middle grouping, which are in turn connected to the rightmost grouping

resulting networks were used to study the dynamics of recurrent spiking networks in general, but also show the capability for self sustained activity and dynamic memory.

3.2.5 Synfire Braids

Conceptually similar to synfire chains, a synfire braid (Gerstein et al., 2012) is a feedforward network of pathways through the network which all begin at different neuronal groupings and lead to the same grouping of neurons (see Figure 3.2). All pathways through the network to a given grouping have the same total delay length.

Similar to synfire chains, synfire braids can be used to study the characteristics of spiking neural networks. Fortunately, some of the techniques used for detecting synfire chains also work for detecting synfire braids; this is because synfire braids are a more general form of synfire chains (Gerstein et al., 2012).

3.2.6 Synfire Rings

A generalisation of synfire chains, synfire rings (Zheng and Triesch, 2014) are synfire chains that begin and end at the same grouping of neurons (see Figure 3.3). Due to the synchronous nature of the synfire chains which make up a ring, any grouping in the ring will receive signals from previous grouping after a fixed time period. In effect, this allows input patterns to become "trapped" in a synfire ring and repeated with a fixed period.

It has been shown (Cabessa and Masulli, 2017) that spiking networks using synfire rings can perform any computation which can be performed by a finite state automata.

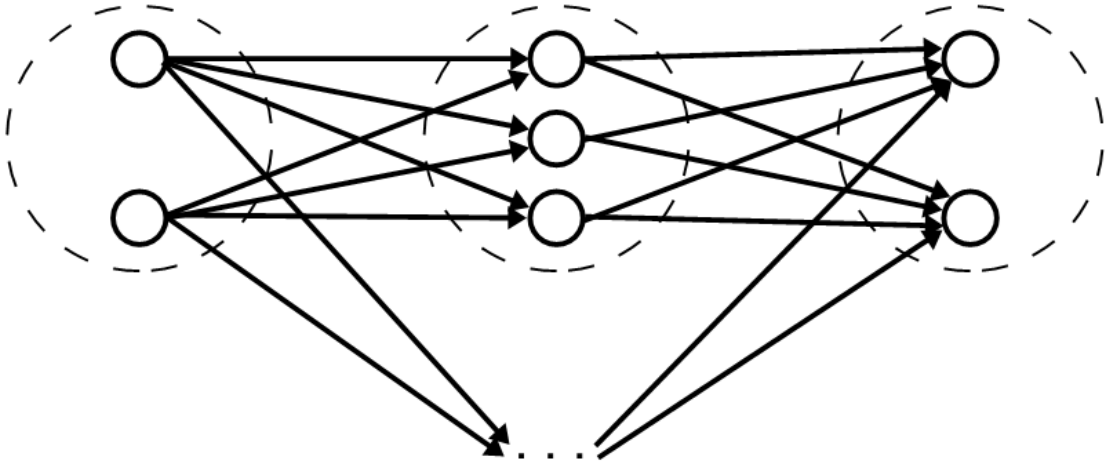


Figure 3.2: Diagram of a synfire braid. Neurons in the leftmost grouping are connected to neurons in the middle grouping, which are in turn connected to the rightmost grouping. Neurons in the leftmost grouping are also connected to the rightmost grouping either directly or through a series of one or more groupings (represented by the ellipsis)

See Section 3.2.9 for an overview of bistable memory loops, which are very similar in concept to synfire rings (except that they do not necessarily make the assumption that the delays are synchronous).

Of particular interest to this thesis is an idea presented in (Kompass, 2004), which discusses some strategies for allowing synfire rings to interact with one another in a synchronous fashion. The article describes how the groupings in synfire rings found in biological brains fire at a rate which is an integer multiple of some global oscillation period (measured to be roughly in the range of 4-5ms). In Chapter 4 of this thesis, the technique which is proposed for synchronizing input pulses to neuronal groupings makes use of a highly similar global oscillation period. The work done by Kompass not only lends support to the idea that such a technique could be effective, but also that there may be some biological precedent to the idea.

3.2.7 Analysis of Spiking Neural Network Dynamics

As described above, it can be difficult to interpret how spiking neural networks work due to the complex dynamics at work. A number of techniques have been developed to investigate the inner workings of SNNs and shed light on their behaviours. Some

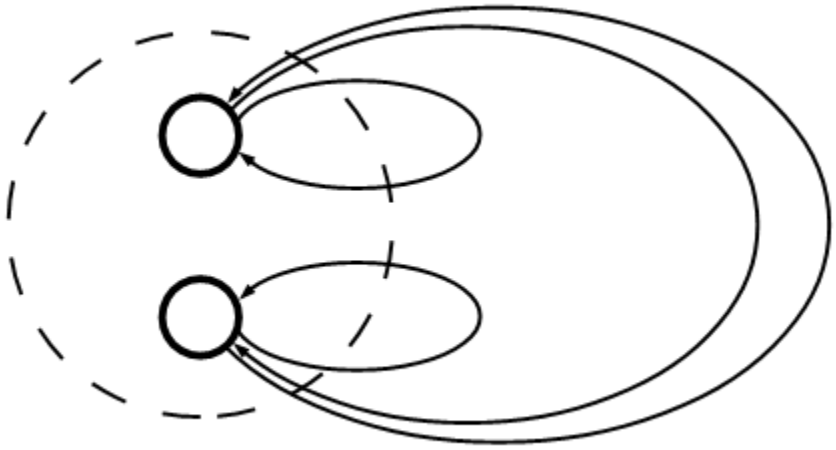


Figure 3.3: Diagram of a synfire ring. Neurons in the grouping are recurrently connected to themselves and other neurons in the grouping

of these techniques are mentioned in this section.

Probability Distributions

Several useful properties of SNNs can be approximated using probability distributions. The research conducted in (Jeanson, 2013) included derivations of distributions used to measure a variety of SNN properties. These included distributions for estimating the total number of neurons firing at a given time (based on the number of neurons in the network, the average degree of the network, the threshold of neurons in the network, and the initial amount of activity) and the number of possible patterns which a network can store with dynamic memory.

Principal Component Analysis

When given the spiking behaviour of neurons over time from a SNN, principal component analysis (PCA) can be used to help identify the individual spiking patterns of neuronal groupings which combine to form the overall behaviour (Chapin and Nicolelis, 1999). PCA works by using eigendecomposition to find strongest components which make up an overall matrix. In the case of SNNs, the strongest components may correlate to neuronal groupings.

Sparse Convolutional Coding

Sparse convolutional coding has been used to identify dynamical cell assemblies of neurons (Peter et al., 2016). This approach works by treating the matrix representing neuron spiking activity as a convolution of matrices which represent the firing of dynamical cell assemblies and matrices which represent neurons which do not fire as part of an assembly. A sparse coding technique is then used to find the optimal convolution that describes the overall spiking behaviour of the network. An advantage of this approach is that it is able to identify neurons which fire in groupings in a non-synchronous manner.

Transfer Entropy

Transfer entropy is a mathematical technique which can be used to measure the flow of information between different parts of a network; transfer entropy outperforms a number of other information theory measures for evaluating connectivity in a network (Garofalo et al., 2009). Variations of transfer entropy have also been used to successfully measure connectivity in spiking networks (Ito et al., 2011). In addition to evaluating connectivity, transfer entropy based measures have been used to make observations on the structure of networks generated by evolutionary means (Vasu and Izquierdo, 2017). Vasu and Izquierdo have determined that neurons in embodied spiking networks trained through evolutionary means form functional groupings which each serve to perform a specific task. Their results also show the effectiveness of transfer entropy to the analysis of spiking networks in general.

3.2.8 Polychronous Neuronal Groupings

A Polychronous neuronal grouping (PNG) (Izhikevich, 2006) is a set of neurons whose synapses have delay lengths such that when one subset of the neurons fire (not necessarily at the same time), the signals that they generate propagate towards another subset of neurons in the grouping and all arrive simultaneously due to the differences in synapse delay lengths. This is essentially the same as synfire braids, save that the restriction that neurons/groupings of neurons do not need to fire at the same time for their signals to arrive at the same time. PNGs have been shown to be capable of

efficiently representing patterns using limited training data (Rekabdar et al., 2015).

Work has been done to analyse the effects of PNGs on the theoretical limits of the processing power of SNNs. While a single neuron in an SNN (with parameters based on those found in biological brains) could theoretically have the ability to decode up to roughly ten kilobits of data per second (Cessac et al., 2010), it has been determined that a single neuron would take much longer to decode the same amount of data from a PNG (Cajueiro and Ranhel, 2015). Specifically, under the experimental conditions described in their paper, Cajueiro and Ranhel found a single neuron to be capable of decoding only about 146 bits per second.

As with synfire chains, braids, and dynamical cell assemblies, identifying PNGs can yield valuable information about SNNs. One approach to identifying PNGs (Sun et al., 2014) uses "readout" neurons with a STDP rule applied to their inputs. Over time, the input synapses which result in the readout neuron firing are strengthened, allowing the detection of PNGs. Another approach to detecting PNGs (Chrol-Cannon et al., 2017) works by assigning codes to neurons in the network. Signals reference the codes of neurons from which they originated, so that when a neuron fires it can track the neurons which contributed to its firing.

3.2.9 Bistable Memory Loops

A number of techniques have been proposed for storing memories in spiking neural networks. Some techniques, such as STDP (Song et al., 2000) encode memory using structural changes to the network or its topology. For example, structural changes can take the form of changing the weights or delay lengths of synapses. While encoding techniques based on making structural changes to the network are capable of forming memories analogous to a human's long-term memory, they cannot easily be used to encode memories quickly. For encoding memories which must be used on a relatively short time scale (akin to a human's short term memory or working memory), a different technique may be used.

Bistable memory loops (Ranhel et al., 2011) take advantage of the spiking properties of SNNs to encode memory. The basic idea behind bistable memory is that pools of polychronous neuronal groupings are connected in a cycle so that one grouping's activation causes a series of other groupings to fire until one of the subsequent

groupings causes the initial grouping to fire again. This cycle of groupings is referred to as a bistable memory unit (BML), since all groupings in the loop will periodically fire once one of them is activated.

By adding a mechanism to inhibit the firing of one or more neuronal groupings in a BML, the loop in question may be activated or inhibited. With this, the loop may be used to represent one of two possible states: the first (on) state is represented by the presence of signals traversing the loop activating groupings, while the second (off) state is represented by a lack of activity. Together, the two states allowed by BMLs allow a simple form of memory to exist in a spiking network without needing to change the structure of the network where state is encoded using the presence or absence of signals rather than using the structure of the network itself.

BMLs allow for spiking networks to be trained to perform sophisticated tasks, such as counting (Ranhel et al., 2011). BMLs can be combined with PNGs which perform basic logic functions to form a technique known as neuronal assembly computing.

3.3 Neuronal Assembly Computing

A framework (Ranhel, 2013) for computing finite state machines (FSM) using SNNs exists. It incorporates the ability to produce networks of standard logic gates such as AND, OR, NOT, and NAND along with basic state representation implemented using bistable memory loops. The author of the framework includes instructions detailing how to convert a finite state machine diagram into a network of neuronal groupings capable of performing the computations described by the FSM.

An advantage of constructing SNN topologies using the technique described in (Ranhel, 2012a) is that the network does not act as a black box, but rather exhibits behaviours described by the user explicitly. This is useful, as it allows for a bottom-up approach to topology design and a better understanding of why the network behaves as it does.

As the number of neural groupings required to model an FSM using neuronal assembly computing (NAC) grows, so to does the complexity of the design of the network (Ranhel, 2012b). Modularity in the SNN may be used to compensate for this (Caelli et al., 1999). One of the contributions of this thesis is a technique for merging modular neuronal groupings into NAC networks in a recurrent, nested manner.

Ranhel mentions two challenges which are faced when working with neuronal assembly computing: phase shift of inputs and multiple triggering events (Ranhel, 2012a). Both challenges were encountered in the research carried out for this thesis, so they are described here.

3.3.1 Phase Shift

Since BMLs store data as pulses constantly travelling between two or more neural assemblies, they can only be properly "read" or "written to" when pulses are at one of the assemblies. If the input to the BML arrives while the BML's pulses are between assemblies, the inputs are considered out of phase. This is because if one were to "read" to a BML while its pulses were between assemblies, it would not be able to give its actual value. Likewise, "writing to" a BML while its signals are in transit would also have problems. Clearing or "zeroing" the BML would have no effect, since the inhibitory signal(s) must arrive at an assembly at the same time as the signal to be cleared, and "setting" a BML while it has signals between assemblies would result in the loop containing more than one set of pulses (which would make clearing it much more complicated).

Ranhel mentions a number of potential solutions to the phase shift issue. One solution is to have a number of assemblies which are each designed to process the input, and each assembly would be acting with a slightly different phase shift. In this way, an input signal would trigger whichever of the associated assemblies is closest to being in phase with the input. Another solution proposed by Ranhel is to use internal oscillators to force input signals to match the phase of BMLs in the network. This latter solution is what was used in this thesis (see Chapter 4, Section 4.6).

3.3.2 Multiplicity of Triggering Events

The other challenge mentioned by Ranhel, multiplicity of triggering events, occurs when a grouping fires repeatedly at a rate faster than its signals (whether excitatory or inhibitory) can be processed by other parts of the network. This can cause a number of issues, such as the issue described in Section 3.3.1 above where a BML has more than one set of signals looping simultaneously.

Ranhel suggests that some kind of assembly could be used to inhibit assemblies

which fire too rapidly. He also suggests the possibility of assemblies which regulate the flow of signals from multi-excitatory or multi-inhibitory assemblies to other assemblies in the network.

3.3.3 Neuronal Assembly Computing and Spike-Time Dependant Plasticity

Work has been done to examine the effects of STDP on NAC networks (Eskandari et al., 2016a). Eskandari et al. found that applying an unsupervised STDP learning rule to a NAC network resulted in an improvement in the overall performance of the network; this is to say that assemblies triggered more strongly (more individual neurons would fire in assemblies which were in an overall firing state) and at a higher rate. These results suggest that STDP may be a useful tool for fine-tuning NAC networks and potentially could be used to improve robustness of NAC networks.

3.4 Summary

This chapter examines research related to the general topic of modular machine learning as well as works which examine the architecture and topologies of SNNs.

Ensemble methods may be used to combine machine learning techniques which perform the same function, but do not address how modules with different functions could be combined. Mixtures of experts are more flexible, but due to the encoding and decoding necessary for them to function they are not ideal for combining modules with a recurrent architecture. Networks of networks and connectionist glue show more promise for spiking neural networks specifically, although it may be difficult to apply them in practical situations. Chapter 4 presents a technique which potentially allows the network of networks approach to be applied to recurrent modular SNNs.

A number of learning techniques for SNNs, both supervised and unsupervised, were examined. These techniques are viable for a wide range of applications, but they tend to focus on training networks in a monolithic manner that does not easily allow additions once the training has been completed. Reservoir computing techniques are used to leverage the complex dynamics of recurrent neural networks without understanding how the internal structure of the reservoir functions. Similar to the more traditional learning techniques, reservoir computing can be used to develop

networks with sophisticated behaviours; scalability to larger problems remains an issue.

Various types of dynamical cell assemblies, from simple synfire chains to more general polychronous neuronal groupings were discussed. These assemblies can provide the backbone for building functional SNNs from the bottom up, and provide a useful tool for top-down analysis of trained SNNs. The technique proposed in Chapter 4 could be thought of as a generalization of synfire chains or braids to function in nested, recurrent modules. Other techniques for the analysis of activity and dynamics of SNNs are also discussed.

Finally, an overview of bistable memory loops and their ability to store short-term memory, as well as their use in the field of neuronal assembly computation, is provided. Some of the example modules and networks in Chapter 5 contain what are functionally BMLs. In addition, the networks in Chapter 5 are effectively NAC networks and demonstrate the utility of the research carried out in this thesis to that field.

Chapter 4

Synfire Circuits

4.1 Introduction

The main goal of the research done in this thesis was to develop a technique that sets the delay lengths of synapses which connect two or more spiking neuronal groupings together such that all signals to a given grouping arrive simultaneously; the network of groupings may contain cycles of groupings as well as groupings which contain internal cycles. In this thesis, the resulting network is referred to as a synfire circuit. The starting and end points of inter-grouping synapses must be provided.

This chapter beings with a formal problem definition in Section 4.2. Next, Section 4.3 describes the spiking neuron model used in this thesis and Section 4.4 presents the algorithm used to simulate a network of spiking neurons. Section 4.5 describes the mathematical concepts, or axioms, which form the foundation of the algorithm used to generate synfire circuits. Section 4.5 also provides proofs that if the axioms are true then the problem will be solved and an algorithm specification for generating synfire circuits. Section 4.6 addresses a common problem which arose when testing the algorithm and provides a solution along with an algorithm specification describing how to solve it.

4.2 Problem Definition and Scope

The problem to be solved in this chapter can be expressed with the tuple $P = (G, I, IC, OC)$, where:

- G is a set of neuronal groupings with particular properties called components. A component is defined by the tuple $C = (N, S, In, Out, Period, Delay, Periods\_Per\_Computation)$, where:
  - N is a set of neurons contained in the component

  - S is a set of intra-component synapses which connect between neurons in N
  - In is a set of neurons which are marked as inputs to the component
  - Out is a set of neurons which are marked as outputs to the component
  - $Period$ is the length of time that passes between one or more of the component's output neurons firing. An output does not need to fire every period, but may only fire at integer multiples of the period.
  - $Delay$ is the length of time that it takes for the component to generate an output signal after an input is received by an input node
  - $Periods\_Per\_Computation$ is a computed property of the component used only in Section 4.6 (see Section 4.6 for the algorithm used to compute $Periods\_Per\_Computation$ and information on how it is used)

- I is a set of inter-component synapses whose delay lengths must be assigned values
- IC is a set of components which are marked as inputs to the network of groupings
- OC is a set of components which are marked as outputs to the network of groupings

In order to solve P, delay lengths must be found for all synapses in I such that signals arriving at any component in G arrive at integer multiples of some global period (see Section 4.5).

Together, G and I define a network of components and inter-component synapses, (G,I).

There are two types of component referred to in this thesis: feed-forward components and cycle-containing components. A feed-forward component is a component which does not contain any cycles of synapses. A cycle-containing component is any component which does contain a cycle of synapses.

Finally, it should be noted that any network generated by the algorithm described in this chapter is usable as a component itself, and has the $Period$, $Delay$, and $Periods\_Per\_Computation$ attributes calculated automatically.

4.3 Neuronal Model

In the research reported here, a modified version of the neuronal model described in (Jeanson, 2013) was used. As in (Jeanson, 2013), time is modelled as discrete to simplify the math in the algorithm and to improve performance.

Since the research carried out in this thesis is concerned with ensuring that signals always arrive precisely simultaneously to the input neurons of any given neuronal grouping, it was decided to always use values of zero for both refractory period and leak time; this means that the equations and algorithm in this chapter would likely need to be adapted somewhat if these variables were used (see Chapter 6).

The modified neuronal model used in this thesis is as follows:

$$a_i(t_n) = \theta[\sum_j w_{ij} a_j(t_n - d_{ij})], \tag{4.1}$$

where $a_i(t_n)$ is the activation state of neuron i at time step n, j is an index over all neurons with outgoing synapses leading to neuron i, w_{ij} is the weight of the synapse from neuron j to neuron i, and d_{ij} is the delay time of the synapse from neuron j to neuron i. θ is the activation function for neuron i, which returns 1 if the parameter is greater than or equal to neuron i's activation threshold and 0 otherwise.

4.4 Spiking Neural Network Simulation Algorithm

Algorithm 1 (page 45) specifies how spiking neural networks are simulated in this thesis. Note that all neurons are updated synchronously.

4.5 Mathematical Foundation

The algorithm described in this chapter generates a set of constrained equations which are used to set the delay lengths of inter-component synapses. This section formally describes these equations as a series of axioms (a.k.a. design properties or constraints) and explains how they are used to combine components with associated theorems.

Definition 1 (Path)**.** *A path is a chain of connected components and the synapses which connect them. A path may form a cycle from beginning to end. In the implementation used in this thesis, a path is represented as a list of tuples which each*

contain a synapse and the component that the synapse originates from.

A path may end at any component and must begin either at an input to the network as a whole (inputs to the network are represented as components in this thesis) or at an entry point to a cycle of components.

A path has an effective delay length (referred to as the path's length) which is calculated with the following summation:

$$\sum_i s_i + k_i c_i, \tag{4.2}$$

where i is an index over all tuples in the path, s_i is the delay length of the i^{th} tuple's synapse, k_i is an integer greater than or equal to 1 that is used to specify how much to scale the i^{th} tuple's component by (see Axiom 3), and c_i is the i^{th} tuple's component's delay attribute.

It should be noted that although numerous paths through the network are possible, a defined path is one which has been detected and represented by the algorithm. Thus, there is a distinction between a defined path and a possible path.

In this thesis, a path is represented as a chain of components connected by arrows. For example, A→B→C represents a path which contains the tuples (component A, synapse A→B) and (component B, synapse B→C). Component C is not actually a part of the path, but it is shown in the path notation so that it is obvious where the path leads. Similarly, a cyclical path might be expressed as M→N→O→M. In this example, the path would contain the tuples (component M, synapse M→N), (component N, synapse N→O), and (component O, synapse O→M). Component M only appears once in the path, but it is included again at the end of the path notation to indicate that the path leads back to component M.

Now that components and paths have been defined, it is possible to state the first of the axioms which describe how to set constraints for delay lengths.

Axiom 1. *All defined paths leading to a particular component must have the same effective delay length.*

Axiom 1 essentially describes a synfire chain ending in a particular component. If Axiom 1 holds, then the following theorem can be used to reduce the number of total constraints on the network.

Theorem 1. *If Axiom 1 holds for all components in a feed-forward network of feed-forward components, and each component has for all of its input synapses at least one path defined which ends at the synapse, then all possible paths to that component will have the same effective length.*

Proof. This theorem may be proven by induction.

Base Case: All possible paths to a component contain only a single input component. In this case, all possible paths to the component will be defined, since there must be a path defined for all input synapses to a component and in this case there is only one possible path for each input synapse. If Axiom 1 is taken to be true for all components, then all possible paths to the component have the same length.

Induction: Consider any pair of possible paths to component Z. Let X be the component which leads to Z in the first path and let Y be the component which leads to Z in the second path. If X and Y are not the same component, then they must connect to Z via different synapses, so at least one path will be defined for each synapse. If X and Y are the same component, then we assume from induction that all possible paths leading to X/Y have the same length. All possible paths from X/Y must traverse a synapse from X/Y to Z. Since a path is defined for all input synapses to Z, then all possible paths which end with a synapse from X/Y to Z are defined and have the same length. From Axiom 1 and induction, all possible paths which end at component X/Y have the same length as the defined paths ending at X/Y. Therefore, all possible paths leading to component Z have the same length. □

In practice, Theorem 1 means that when defining a path to a given component, one does not need to worry about which possible path is taken when traversing backwards through the network from the given component, since any possible path will have the same length as the one defined.

For example, consider the network in Figure 4.1. Since component A has only a single input synapse, only one path will be defined which leads to A. However, component B has two incoming synapses. This means that two paths will be defined which lead to B, path C→B and path E→D→B. Since they are both defined, C→B and E→D→B will be required to have the same lengths. This means that it makes no difference which incoming synapse to component B is used to define the path leading to component A.

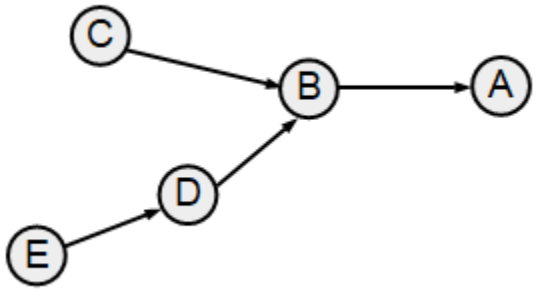


Figure 4.1: Example of a network with four possible paths: C→B, E→D→B, C→B→A and E→D→B→A. Only three of these paths would be defined: C→B, E→D→B and one of C→B→A or E→D→B→A.

If Axiom 1 is followed, then Theorem 1 is all that is needed to ensure that a feed-forward network of feed-forward components is connected in such a way that all components will receive their input signals simultaneously. Next, consider a network of feed-forward components with one or more cycles of components. Firstly, some definitions:

Definition 2 (Global Period)**.** *The global period is a variable which must be solved for by the constraint solver. It is used to synchronise the firing rates of components and cyclical paths. The delay times of all cycles (cycles of components and cycles within components) in the network are proportional to the global period. If the network is later used as a component, the resulting component's period will have a value equal to the original network's global period.*

Definition 3 (External Input)**.** *An external input is a synapse which leads into a cycle but is not part of the cycle itself.*

Definition 4 (Periodic Section)**.** *A cycle is composed of one or more periodic sections. A periodic section is the part of a cycle which is between a a pair of points (neurons or components) of the cycle which have external inputs. If the cycle has either one or no external inputs, then the entire cycle is considered a periodic section.*

Figure 4.2 shows an example of a network containing a cyclical path of components, A→B→C→D→A. Component E has a synapse which leads to component

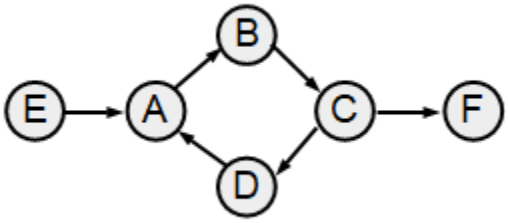


Figure 4.2: Example of a network containing a cycle with one periodic section.

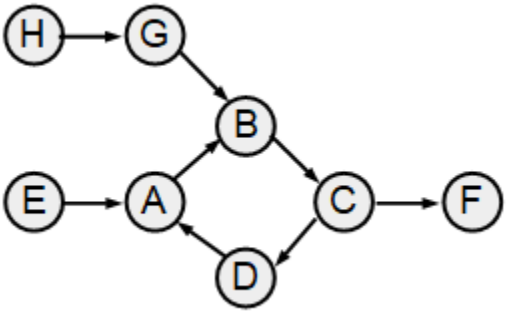


Figure 4.3: Example of a network containing a cycle with two periodic sections.

A, this synapse is an example of an external input to cycle A→B→C→D→A. Since A→B→C→D→A can be traversed all the way back to A before encountering another external input, cycle A→B→C→D→A has only a single periodic section (which is the entirety of the cycle).

Figure 4.3 shows the network from Figure 4.2 with the addition of component G and a synapse leading from G to component B. This synapse is an external input to cycle A→B→C→D→A for the same reason that synapse E→A is. As a result, cycle A→B→C→D→A has two periodic sections: A→B and B→C→D→A.

Presented here is an axiom which is intended to ensure that components in the cycle-containing network of feed-forward components continue to receive their inputs simultaneously:

Axiom 2. *All periodic sections in a cycle of components must be defined as paths which have length equal to the global period.*

Theorem 2. *If Axiom 1 and Axiom 2 both hold, then Theorem 1 is true even in*

a network of feed-forward components which contains one or more cycles and any signals which pass through a particular component will arrive simultaneously with other signals arriving at that component.

Proof. Consider a cycle of components with at least one external input. Let X be a component in the cycle with an external input leading to it. Since Axiom 1 holds, any path leading to X must have the same length as any other path leading to X. This means that the paths leading to X which are not part of X's cycle will have length equal to the length of the path which is a section of X's cycle and leads to X. Since Axiom 2 holds, this means that all paths leading to X will have length equal to the global period.

Now consider an arbitrary component Y and any two paths leading to Y, y1 and y2. By the definition of a path, y1 and y2 must either start at input components or components in cycles which are where external inputs to the cycles lead (such as X).

If y1 and y2 both originate at input components, then Theorem 1 applies normally.

If y1 (or y2) is a path originating at a component in a cycle with an external input (let this component be Z if it is the start of y1 and W if it is the start of y2), then the cycle may "produce" a new signal to send down the path every time a signal outputted by Z/W traverses Z/W's cycle and reaches Z/W again. This will take a length of time equal to the global period times the number of periodic sections in Z/W's cycle.

If y1 originates at a component in a cycle and y2 does not, then y1's length will be equal to the global period plus the distance from the end of y1's cycle to Y. If Axiom 1 is followed then y2's length would be set to the same length, and y1 and y2 would have the same length. This also applies if y2 originates at a component and y2 does not.

Finally, if y1 and y2 both originate at components in cycles (component Z and W, respectively), then both paths will have length equal to the global period plus the length of the rest of the paths (which are not in Z/W's cycles). Z and W will always generate signals with a period equal to some integer multiple of the global period, and those signals will take the same amount of time to reach Y. Therefore, every interval of time equal to the global period (with some constant offset for each component), every path to any component will either deliver a signal to the component or not.

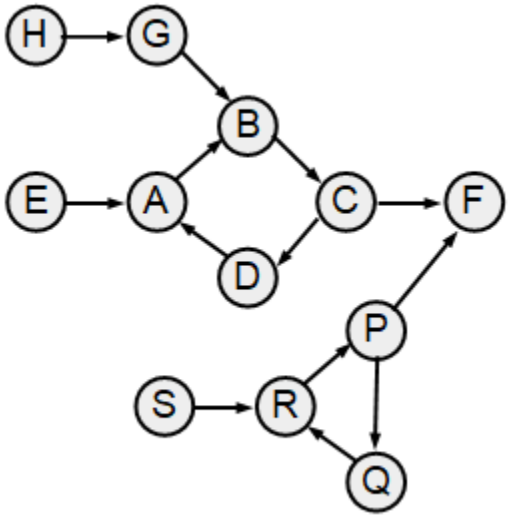


Figure 4.4: Example of a network containing a two cycles. Axiom 2 states that each periodic section in each cycle must have a length equal to the global period.

Components will never receive signals which are out of phase with their offset version of the global period. □

Axiom 2 means that both periodic sections of cycle A→B→C→D→A in Figure 4.3 would need to have the same delay length; the value used for the delay length is the global period. Figure 4.4 shows the same network as the one in Figure 4.3 except with another cycle, P→Q→R→P. Cycle P→Q→R→P has only a single external input, synapse S→R, so cycle P→Q→R→P has only a single periodic section (itself). If Axiom 2 is followed, then the paths A→B, B→C→D→A, and P→Q→R→P all have the same length.

The final obstacle that must be addressed is cycle-containing components. This is done by ensuring that the following axiom is true:

Axiom 3. *All cycle-containing components must be scaled (have the delay lengths of all internal synapses multiplied) by some integer such that the value of the component's period attribute times the scaling factor is equal to the global period. Different components may have different scaling factors.*

It is important to remember certain properties of components here. Specifically, it is important to note that cycle-containing components always treat their delay as

being 0 and that the length of all routes in the component that lead to the component's output neurons are equal to the component's period value (ie, the component will fire with a period equal to its period attribute). Axiom 3, along with Axiom 1 and Axiom 2, allow the following theorem to be formulated:

Theorem 3. *If Axioms 1, 2, and 3 hold, then Theorems 1 and 2 apply to networks of feed-forward or cycle-containing components which may contain cycles.*

Proof. The proof for this theorem is the same as that of Theorem 2, save for the following special case.

If a path, p, from component X to component Z contains a cycle-containing component, Y, then the path will deliver signals to Z simultaneously with other paths which deliver signals to Z (whether they have cycle-containing components in them or not).

Y will be scaled such that its period value times its scale factor is equal to the global period. Since Y is cycle-containing, its delay value is 0. This means that the length of p will contain 0 times Y's scaling factor, which works out to zero. In other words, Y does not contribute to the overall length of p. If all other things are equal between path p and another path, q, from X to Z which does not contain Y then p and q would have the same length. This means that any signal traversing path p will have to travel through Y while a signal traversing q. However, Y produces a signal with a period equal to the global period (specifically, it takes some integer multiple of the global period to produce a signal after receiving a signal from a path leading to Y). Since p and q have the same length, signals produced by Y will always arrive at Z simultaneously with signals which traversed q. □

Theorems 1 through 3 allow the delay lengths of inter-component synapses to be set such that each component in a network (which may contain cycle-containing components and/or cycles of components) has signals arrive at its input neurons simultaneously.

For example, let component G in Figure 4.4 be the cycle-containing component shown in Figure 4.5. If the global period of the network shown in Figurefig:example4 had a global period of 8, then component G would need to have its internal synapses scaled up by a factor of 2 (multiplying the delay length of each internal synapse by

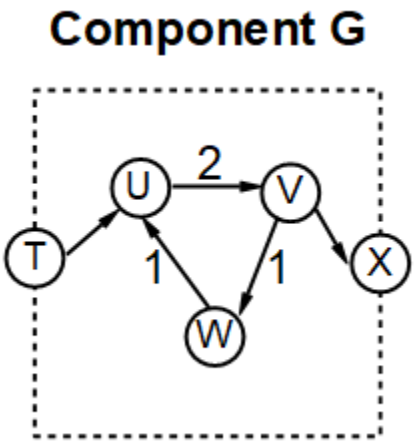


Figure 4.5: Internal structure of example component G. T, U, V, W, and X are neurons. T is an input neuron, and X is an output neuron.

2). The result of setting component G's scale to 2 would result in G's effective period being 8, the same as the global period of the network.

There is one final axiom which allows these networks to be used as components themselves in other networks:

Axiom 4. *All paths which end at any output component must have the same length. If the network contains either cycle-containing components or cycles of components, then this length must equal the global period.*

The first part of Axiom 4 is simple, and exists so that paths containing the network in component form do not have to take into account which internal path through the component a signal traverses. If the network is used as a component, then that component's *delay* would equal the length of paths going to output components in the original network. The second part exists so that the network will fire with a period equal to its global period when it is packaged as a component.

For example, consider the network in Figure 4.4. Let component F be the network's single output component. In order for Axiom 4 to be satisfied, the paths B→C→F and R→P→F would both need to have a length equal to the global period.

In order to construct a network with a synfire circuit topology, Axioms 1 through 4 need to be expressed as constraints on the delay lengths of inter-component synapses; a constraint solver can then be used to solve for the delay lengths.

4.5.1 Algorithm Specification of the Synfire Circuit Algorithm

Algorithm 2 (page 46) is used to solve P. It makes use of two smaller algorithms (helper functions), which are described by Algorithm 3 (page 48) and Algorithm 4 (page 49). The constraint solver used for this thesis was Choco-Solver (Prud'homme et al., 2017). The objective function used for optimization was to find the solution with the minimum possible total delay length for all inter-component synapses.

4.6 Handling Phase Shift of Inputs and Multiplicity of Triggering Events

While the previous section details the theoretical basis and algorithmic construction of synfire circuits, there are two problems which often arise when using synfire circuits in practice. These problems, mentioned in Sections 3.3.1 and 3.3.2 of Chapter 3 and described in (Ranhel, 2012a), are the phase shift of inputs and multiplicity of triggering events.

In the context of the synfire circuit algorithm, phase shift of inputs occurs when inputs arrive at time steps which are not divisible by the network's global period. For example, if a network's global period is 10ms and it has two input components, A and B, which fire at 5ms and 6ms, respectively, and lead to component C, then it is impossible to ensure that those signals arrive at C at the same time.

Multiplicity of triggering events can occur in synfire circuits when components fire at different rates (even though the components are each firing at multiples of the global period). For example, if two components, C and D, lead to a third component E. If C fires every global period and D fires every other global period, and if E performs some computation based on C and D, then it would treat D as having not fired every other iteration of the global period. This can sometimes cause issues.

Fortunately, both of these issues can be solved with the same technique: internal clocks in components and a global clock for the network as a whole. A clock is simply a cycle (a cycle of synapses and neurons for internal clocks, and a cycle of components for the global clock). The clock outputs to a series of logical AND gates which also receive signals from inputs (to either the component or network). Thus, if a signal arrives which is not expected, it is simply filtered out. The complicated part of setting

up clocks is determining how long the delay should be and ensuring that the delay will stay that way (since the constraint solver will reset all inter-component delay lengths).

The task of setting up these clocks is made easier by the fact that a network's global clock will function as an internal clock if the network is packaged as a component. Since feed-forward components have no need for internal clocks, this section first describes the process of adding a global clock to a network of feed-forward components.

If the network is feed-forward, then a global clock is not needed (and if the network were packaged as a component, it would have a $Periods\_Per\_Computation$ value of -1). Otherwise, cycles in the network must be identified and the number of periodic sections in each cycle must be recorded. The lowest common multiple (LCM) of these numbers is then multiplied by the global period to determine the delay for the global clock. Using the LCM ensures that signals will have a chance to traverse the entirety of all cycles before the next batch of input signals is accepted into the network. This solves the multiplicity problem. The phase shift of inputs problem is solved because the clock has a period equal to some multiple of the global period, so signals which enter the network out of phase with the global period will be filtered out.

If the network has cycle-containing components, a similar process is used. Identify the cyclical paths in the network and record the number of periodic sections in each. In addition, record the $Periods\_Per\_Computation$ value of all cycle-containing components. The LCM of all recorded values is the value which should be used for the clock delay.

Once the delay length needed for the global clock is known, the clock must be constructed in a way that Axiom 2 is respected. To do this, make a cyclical path of single-node components. These are referred to as clock components. The number of clock components should be equal to the number period desired for the clock divided by the global period (so if the clock needed to have a period of 3 global periods, then it would have 3 clock components). Next, add a "dummy" component for each clock component. Each of these "dummy" components, referred to as spurs, are connected to a different clock component by an inter-component synapse from the spur to the clock component. The weight of the connecting synapse should be set to

0. Effectively, this adds a number of "fake" external inputs to the cycle which results in the cycle having a period which is effectively longer than the global period.

Finally, the global clock should have an external input from an input component used to start the clock (referred to as the clock input). The last clock component should also have an outgoing synapse which leads to the clock inputs for all *Periods_Per_Computation* cycle-containing components.

4.6.1 Algorithm Specification for Adding Clocks

Algorithm 5 (page 50) specifies how to add a global clock to the network (G,I). Algorithm 6 (page 51) specifies how the *Periods_Per_Computation* value is computed for a network, and is used in Algorithm 5. Unless otherwise stated, any synapses added to the network by these algorithms have a weight of 1 and their delay lengths are arbitrary (since Algorithm 2 is run at the end to set delay lengths of inter-component synapses). Any components which are added by these algorithms contain a single neuron with a threshold value of 1; this neuron is both an input and output neuron (see Section 5.2.1 in Chapter 5 for details).

Algorithm 1 Algorithm Specification for SNN Simulation

Require: $Nw = (N, S)$, the entire network expressed as a set of neurons, N, and a set of synapses, S

Require: *steps*, the number of iterations to simulate

```
1:  procedure SIMULATE(Nw,steps)
2:      while steps > 0 do
    ▷ Find the sum of all input signals arriving at each neuron:
3:          for n ∈ N do
4:              n.inputSignals ← 0
5:              for s ∈ S do
6:                  if s leads to n then
7:                      signal ← s.signals.PopBack() × s.weight
8:                      n.inputSignals ← n.inputSignals + signal
    ▷ Take the output of each neuron's activation function and push it to the front
    of all outgoing synapses' signal arrays:
9:          for n ∈ N do
10:             for s ∈ S do
11:                 if s begins at n then
12:                     s.PushFront(Activate(n))
13:         steps ← steps − 1
14: function ACTIVATE(n)
15:     if n.inputSignals >= n.threshold then
16:         return 1
17:     else
18:         return 0
```

Algorithm 2 Algorithm Specification for Constructing Synfire Circuits

Require: $P = (G, I, IC, OC)$, the problem as defined in Section 4.2

1: **procedure** SolveForDelayLengths(P)
2: $constraints \leftarrow \{\}$
3: **for** component $g \in G$ **do**
4: $constraints$.Append($g.scale >= 1$)

▷ Find the list of cycles that each synapse in I is a part of:

5: **for** synapse $i \in I$ **do**
6: $i.cycles \leftarrow \{\}$
7: $path \leftarrow \{\}$
8: $visited \leftarrow \{\}$
9: FindCycles(i,$path$,i,$visited$) ▷ see Algorithm 3

▷ Define a path which ends at each synapse in I:

10: **for** synapse $i \in I$ **do**
11: $i.paths \leftarrow \{\}$
12: $path \leftarrow \{\}$
13: $nextComponent \leftarrow$the component i leads to
14: $visited \leftarrow \{\}$
15: **for** synapse $s \in nextComponent.inputSynapses$ **do**
16: $visited$.Append(s)
17: FindPath(i,$path$,i,$visited$) ▷ see Algorithm 4

▷ Implement Axiom 1:

18: **for** component $g \in G$ **do**
19: $constraints$.Append(variable v_g)
20: **for** synapse $s \in g.inputs$ **do**
21: $constraints$.Append($s.path.length = v_g$)

▷ Implement Axiom 2:

22: $constraints$.Append(variable $globalPeriod$)
23: **for** synapse $i \in I$ **do**
24: **for** path $c \in i.cycles$ **do**
25: **for** periodic section $p \in c$ **do**
26: $constraints$.Append($p.length = globalPeriod$)

▷ Implement Axiom 3:

for component $g \in G$ which is a cycle-containing component **do**

    $constraints$.Append($g.period \times g.scale = globalPeriod$)

▷ Implement Axiom 4:

$constraints$.Append(variable $outputPathLength$)

if there are any cycle-containing components in G or if any synapse in I has a non-empty *cycles* array **then**

    $constraints$.Append($outputPathLength = globalPeriod$)

for synapse $i \in I$ which leads to a component $\in OC$ **do**

    $constraints$.Append($i.length = outputPathLength$)

▷ find delay lengths for each synapse in I such that *constraints* are satisfied:

Solve($constraints$)

Algorithm 3 Algorithm Specification for finding cycles which contain a given inter-component synapse

Require: *currentSynapse*,*path*,*destinationSynapse*,*visited*

```
1: procedure FindCycles(currentSynapse,path,destinationSynapse,visited)
2:     prevComponent ← the component currentSynapse originates from
3:     path.Append({prevComponent, currentSynapse})
4:     if path is a cycle then
5:         destinationSynapse.cycles.Append(path)
6:         return
7:     visited.Append(prevComponent)
8:     candidates ← {}
9:     for synapse s ∈ prevComponent.inputSynapses do
   ▷ Make sure s does not connect to path in any way
10:        if visited does not contain s then
11:            if no synapse in path has the same starting neuron as s's starting neuron then
12:                if no synapse in path has the same ending neuron as s's ending neuron then
13:                    if s's starting neuron is not a member of any groupings in visited then
14:                        candidates.Append(s)
15:    for synapse s ∈ candidates do
16:        pathCopy ←copy of path
17:        visitedCopy ← copy of visited
18:        visitedCopy.Append(s)
19:        FindCycles(s,pathCopy,destinationSynapse,visitedCopy)
```

Algorithm 4 Algorithm Specification for finding a path which leads to a given inter-component synapse

Require: *currentSynapse*,*path*,*destinationSynapse*,*visited*

1: **procedure** FINDPATH(*currentSynapse*,*path*,*destinationSynapse*,*visited*)
2: *prevComponent* ← the component *currentSynapse* originates from
3: **if** *prevComponent* is part of a cycle **then**
4: *cycle* ←the cycle *prevComponent* is a part of
5: *tailPath* ← {{*prevComponent*, *currentSynapse*}}
6: *subPathInCycle* ← the portion of *cycle* from *prevComponent* backwards to the first external input
7: **for** each segment p of *subPathInCycle* except the first **do**
8: *tailPath*.Append(p)
9: *path*.Append(*tailPath*)
10: *destinationSynapse.paths*.Append(*path*)
11: **return**
12: *path*.Append({*prevComponent*, *currentSynapse*})
13: *visited*.Append(*prevComponent*)
14: *candidates* ← {}
15: **for** synapse $s \in$ *prevComponent.inputSynapses* **do**
16: **if** *visited* does not contain s **then**
17: **if** s's starting neuron is not a member of any groupings in *visited* **then**
18: *candidates*.Append(s)
19: **for** synapse $s \in$ *prevComponent.inputSynapses* **do**
20: *visited*.Append(s)
21: FindPaths(first element in *candidates*,*path*,*destinationSynapse*,*visited*)

Algorithm 5 Algorithm Specification for adding a global clock to the network (G,I)

Require: $P = (G, I, IC, OC)$, the problem as defined in Section 4.2 to have been solved

1: **procedure** AddClock(P)
2: $Periods\_Per\_Computation \leftarrow$ ComputePeriodsPerComputation(G,I)
3: $clockIn \leftarrow$ new component
4: G.Append($clockIn$)
5: IC.Append($clockIn$)
6: $clocks \leftarrow \{\}$
7: **if** $Periods\_Per\_Computation <= 1$ **then**
8: $prevClock \leftarrow clockIn$
9: **for** $1 \rightarrow 2$ **do**
10: $clock_i \leftarrow$ new component
11: G.Append($clocks_i$)
12: $clocks$.Append($clocks_i$)
13: $clockSynapse \leftarrow$ new synapse from $prevClock \rightarrow clock_i$
14: I.Append($clockSynapse$)
15: $prevClock \leftarrow clock_i$
16: $clockSynapse \leftarrow$ new synapse from $prevClock \rightarrow 1^{st}$ element in $clocks$
17: I.Append($clockSynapse$)
18: **else**
19: $prevClock \leftarrow clockIn$
20: **for** $1 \rightarrow Periods\_Per\_Computation$ **do**
21: $clock_i \leftarrow$ new component
22: G.Append($clocks_i$)
23: $clocks$.Append($clocks_i$)
24: $clockSynapse \leftarrow$ new synapse from $prevClock \rightarrow clock_i$
25: I.Append($clockSynapse$)
26: $spur \leftarrow$ new component
27: G.Append($spur$)
28: $spurSynapse \leftarrow$ new synapse from $spur \rightarrow clock_i$ with $weight = 0$
29: I.Append($spurSynapse$)
30: $prevClock \leftarrow clock_i$
31: $clockSynapse \leftarrow$ new synapse from $prevClock \rightarrow 1^{st}$ element in $clocks$

32: I.Append($clockSynapse$)
33: **for** each non-clock component $ic \in IC$ **do**
34: $and_i \leftarrow$ new AND component ▷ see Chapter 5
35: $fromClockIn \leftarrow$ new synapse from $clockIn \rightarrow and_i$'s first input
36: $fromOriginalIn \leftarrow$ new synapse from $ic \rightarrow and_i$'s second input
37: I.Append($fromClockIn$)
38: I.Append($fromOriginalIn$)
39: $fromAnd \leftarrow$ new synapse from $and_i \rightarrow$ whichever component inputs ic used to lead to
40: I.Append($fromAnd$)
41: I.Remove(synapse from $ic \rightarrow$ whichever component inputs ic used to lead to)
42: **for** component $g \in G$ which was not added in this procedure **do**
43: $toInternalClock \leftarrow$ new synapse from $clockIn \rightarrow$ last element in g's clocks array
44: I.Append($fromClockIn$)
45: SolveForDelayLengths(P)

Algorithm 6 Algorithm Specification for computing the $Periods\_Per\_Computation$ value of the network (G,I)

Require: G, the components in the network (G,I)
Require: I, the inter-component synapses in the network (G,I)
1: **function** ComputePeriodsPerComputation(G,I)
2: $periodicSectionsPerCycle \leftarrow \{\}$
3: **for** synapse $i \in I$ **do**
4: **for** path $p \in i.cycles$ **do**
5: $periodicSectionsPerCycle$.Append(number of periodic sections in p)
6: **for** component $g \in G$ **do**
7: $periodicSectionsPerCycle$.Append($g.Periods\_Per\_Computation$)
8: **return** LowestCommonMultiple($periodicSectionsPerCycle$)

Chapter 5

Experiments

5.1 Introduction

The techniques described in Chapter 4 for merging spiking neuronal assemblies into more complex networks were tested using a series of experiments. In these experiments, components were designed to simulate digital logic functions. The components were combined by specifying inter-component synapses to connect the components, but not how long the delay lengths of these inter-component synapses should be. The algorithms described in Chapter 4 were then used to automatically determine the appropriate delay lengths for the inter-component synapses.

Section 5.2 describes the basic components which were used as building blocks for constructing more complicated components and networks.

Sections 5.3 through 5.8 show a number of networks which were constructed to test the synfire circuit algorithm. In these sections, inter-component synapses are always assumed to have a weight of 1 and a delay length which is set by the algorithm. Network diagrams for networks which have global clocks do not include the clocks.

In the diagrams throughout this chapter, a dotted box is used to outline the boundaries of a network. Above the dotted box is the network's name and the symbol used to represent it as a component in other networks.

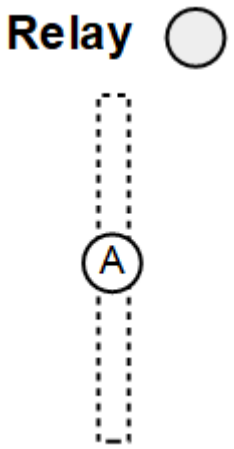
5.2 Basic Components

Almost all of the components used in these experiments were created by merging more basic components together. The exceptions are the base level components described below. Note that unless otherwise specified, it is assumed that input synapses to the components will have weights of 1. All of the basic components are feed-forward networks. It is worth noting that three of the basic components presented below (the AND, OR, and NOT components) form a functionally complete set of boolean

operations. This implies that if the synfire circuit algorithm works as intended, then all possible boolean expressions may be implemented as SNNs constructed as synfire circuits using a combination of these components.

5.2.1 Relay Component

The most basic component used for the experiments in this thesis is called a relay component. It contains only a single neuron, A, with a threshold value of 1; this neuron is both an input and output neuron (i.e., A $\in C.In$ and A $\in C.Out$). Figure 5.1 shows a diagram of a relay component.



Neuron	Threshold
A	1

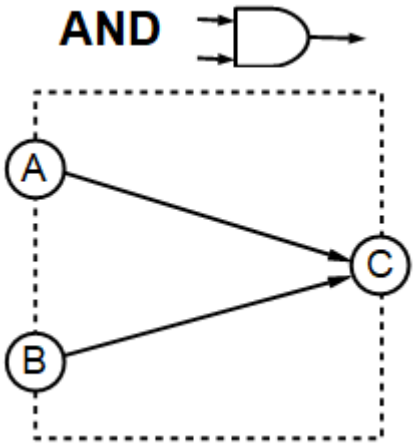
Figure 5.1: Relay component diagram

The relay component is mainly used as an input or output component in other networks. It is called a relay component because it fires any time it receives an input signal and thus acts as a relay. It is useful because it allows more flexibility when setting constraints on synapse delay lengths; synapses leading to the relay may be adjusted separately from those which lead away from it.

5.2.2 AND Gate

The second basic component which was used is one which performs a boolean AND operation (see Figure 5.2). This component uses three neurons: Two neurons (A and B) are used as inputs and one (C) is used for the actual computation and as an

output. Neuron C has a threshold of 2, meaning that it will fire if and only if both of its inputs are 1.



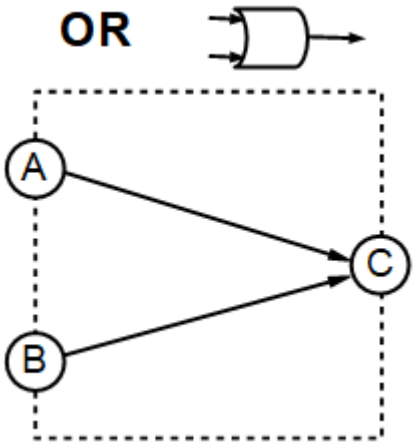
Neuron	Threshold
A	1
B	1
C	2

Synapse	Weight	Delay
A→C	1	1
B→C	1	1

Figure 5.2: AND component diagram

5.2.3 OR Gate

The third component performs a boolean OR operation (see Figure 5.3). Its design is the same as that of the AND component, except that the neuron which performs the computation, C, has a threshold of 1. This means that if either of the inputs sends a signal then the component will return an output of 1.



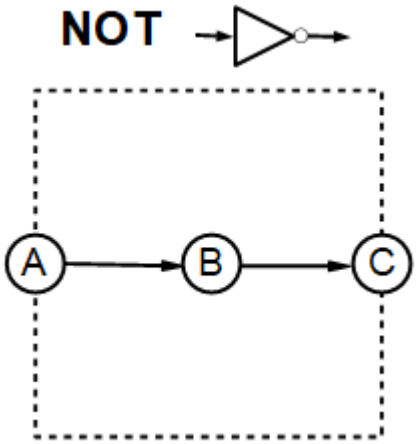
Neuron	Threshold
A	1
B	1
C	1

Synapse	Weight	Delay
A→C	1	1
B→C	1	1

Figure 5.3: OR component diagram

5.2.4 NOT Gate

The NOT gate component (see Figure 5.4) requires two neurons to perform its computation, in addition to an input neuron. The first of the neurons used for computation, B, has a threshold of 1 and a synapse with weight -1 leading to the second computation neuron. The second computation neuron, C, has a threshold of 0, meaning that it will fire constantly so long as it does not receive an inhibitory signal. This results in a component which returns 0 when it receives an input and 1 otherwise.



Neuron	Threshold
A	1
B	1
C	0

Synapse	Weight	Delay
A→B	1	1
B→C	-1	1

Figure 5.4: NOT component diagram

5.3 XOR

The first real test of the synfire circuit algorithm was to use it to construct a network which performs an XOR operation (see Figure 5.5). In this experiment a number of AND and NOT gates were combined into a larger feed-forward network. No global or internal clocks were used.

The combined network worked correctly. The network takes 12 time steps to produce an output signal. Figure 5.6 shows the result of all possible input combinations to the XOR network (black dots indicate that the neuron is firing for a given time step, white space indicates that the neuron is not firing). This suggests that the algorithm correctly implements Axiom 1 and lends empirical evidence to the claim made in Theorem 1.

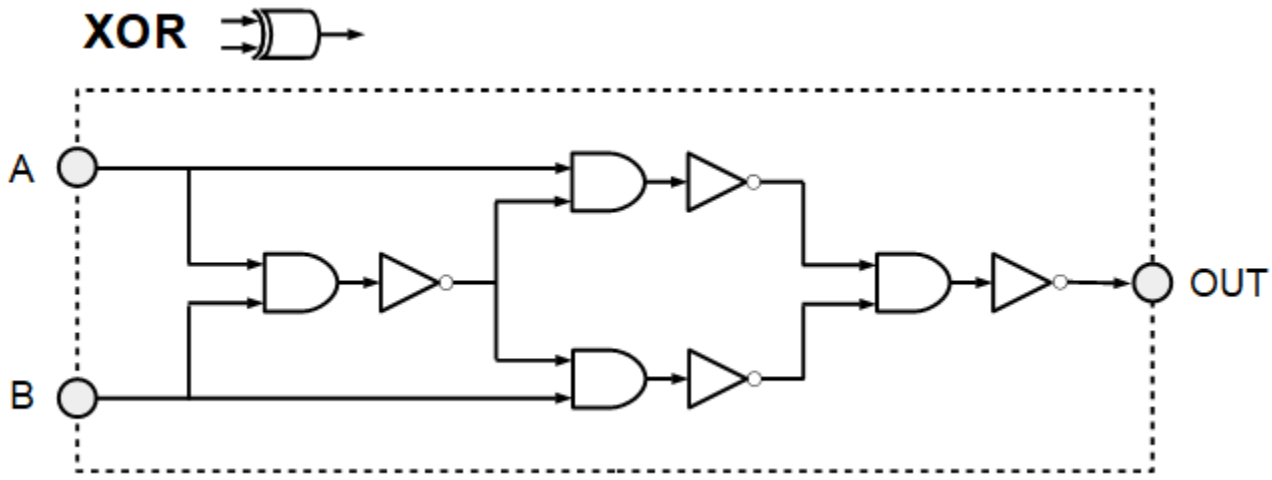


Figure 5.5: XOR component diagram

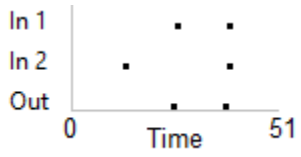


Figure 5.6: Spike plot of input and output neurons for XOR network

5.4 Combined AND/OR/NOT Gate

A component which combined the functionality of the AND, OR, and NOT gates was constructed. This component, called ANDOR, was needed to allow the flip-flop component to function properly (see Section 5.5 for more details). The component takes three inputs. It performs an OR operation on the first and second input, and performs an AND operation on the result and the inverse of the third input. A diagram of the ANDOR network is shown in Figure 5.7. Like the XOR network, the ANDOR network is a feed-forward network of feed-forward components.

Since the ANDOR component is feed-forward like XOR, and it is used in Section 5.5 as part of the flip-flop network/component, results are not shown here. Table 5.1 presents the constraints which were generated for the ANDOR component by the synfire circuit algorithm; this is to provide an example of how constraints are formulated in a practical network. In the table, variables are represented by a "V" with a subscript identifying the variable. Section 5.5 includes a similar example of

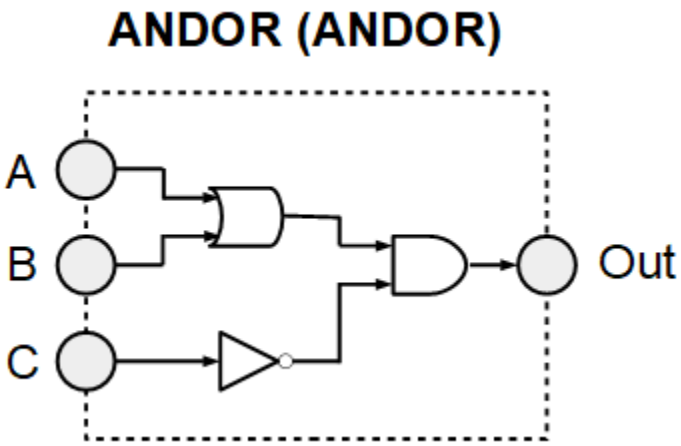


Figure 5.7: ANDOR component diagram

constraints for a network containing cycles of components, and Section 5.7.1 includes an example of constraints for a network containing cycles of components and cycle-containing components.

Path	Length of Path
A→OR	V_1
B→OR	V_1
A→OR→AND	V_2
C→NOT→AND	V_2

Table 5.1: Constraints for the ANDOR circuit

5.5 Flip-Flop

The next test for the algorithm was to use it to create a circuit which has state. The algorithm was used to combine a number of feed-forward components into a network which acts as a flip-flop that can store one bit of information (see Figure 5.8). This network implements state using a cycle of components similar to the bistable memory loops discussed in Section 3.2.9 in Chapter 3.

The design of the flip-flop was a simple AND-OR flip-flop. It has two inputs, S (set) and R (reset), and two outputs, Q and P (the inverse of Q). The basic premise of the circuit is that if S is 1 or the circuit loop between the AND and OR gates is 1 and R is 0 then Q returns a value of 1, otherwise Q returns 0. The constraints

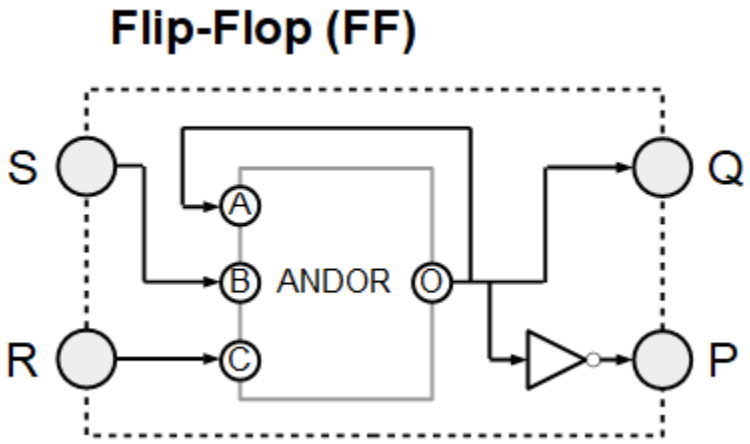


Figure 5.8: Flip-flop component diagram

generated by the synfire circuit algorithm for the flip-flop network (before adding a global clock, see below) are shown in Table 5.2. Note that there are some paths whose lengths must equal more than one variable. In this case, the path's length and the two (or more) variables must have equal values.

Path	Length of Path
S→ANDOR	V_1
ANDOR→ANDOR	V_1, $V_{GlobalPeriod}$
ANDOR→Q	V_{Output}, $V_{GlobalPeriod}$
ANDOR→NOT→P	V_{Output}, $V_{GlobalPeriod}$

Table 5.2: Constraints for the Flip-flop circuit

The resulting network worked well enough so long as the inputs always arrived at exactly the right times. However, if signals arrived out of phase with the signal cycling between the AND and OR gates, then the network could end up storing more than one value. This proved especially problematic when using the flip-flop network as a component in larger networks where the multiple excitation problem described in Section 3.3.2 of Chapter 3 frequently resulted in flip-flops which stored more than a single signal and thus having an uncertain value.

To mitigate this problem, an internal clock was added to the flip-flop component (as described in Section 4.6 in Chapter 4). However, this resulted in another problem. The cycle formed by the AND and OR gates in the flip-flop has two periodic sections. The S input is an external input to the cycle in the periodic section from OR to

AND. The R input is an external input to the cycle in the periodic section from AND to OR. Since each periodic section has a delay length equal to the global period of the flip-flop network, signals from the R input have to arrive one global period after signals from the S input. This causes a problem because the global clock for the flip-flop has two periodic sections; inputs are only accepted into the flip-flop network every two global periods. This means that either S or R inputs are always filtered out.

Combining the AND, OR, and NOT gates into a single component (the ANDOR component, see Section 5.4) solves this problem. Inputs to the flip-flop network go directly to the ANDOR component, which loops back to itself. Since the synfire circuit algorithm does not change the internal structure of components (other than potentially scaling them), the ANDOR component forms a cycle with itself with only a single periodic section and the S and R inputs are accepted at the same time.

With these changes, the flip-flop network behaves correctly (see Figure 5.9) and can be used as a component in other networks. While there were difficulties involved in getting the inputs to arrive at the same time, these difficulties were caused as a direct result of trying to have the network only contain a single signal at a time. Despite these difficulties, the synfire circuit algorithm was able to ensure that components always received input signals after some integer multiple of the network's global period. The results of this experiment suggest that the algorithm correctly implements Axiom 2 and provides some limited empirical evidence of the claims made in Theorem 2.

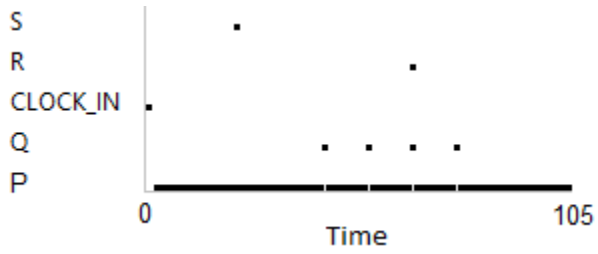


Figure 5.9: Spike plot of input and output neurons of flip-flop network

Figure 5.10 shows a spike plot for all neurons in the flip-flop network. While this plot was recorded, the flip-flop was initially in the "off" state. After 22 time steps, an input signal was injected into input S (putting the network in an "on" state). At this

point, an increase in spiking activity appears and maintains a repeating pattern which can be seen in the plot. This repeating behaviour is consistent with the behaviour observed by Jeanson in (Jeanson, 2013) which corresponded to dynamic memory. After 66 total time steps, a signal was injected into the network through input R and the repeating spiking behaviour ceases.

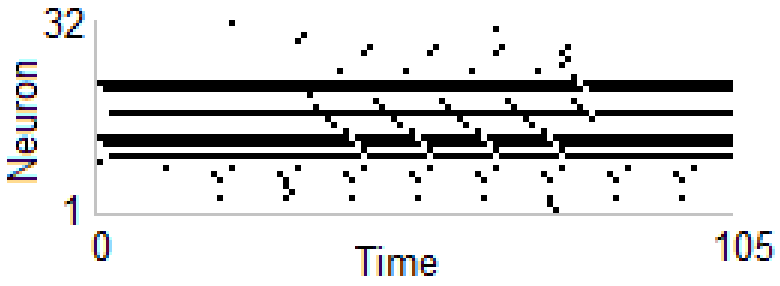


Figure 5.10: Spike plot of all neurons in the flip-flop network

5.5.1 Detailed Analysis of the Flip-Flop's Spike Plot

Some notable features can be observed in the spike plot in Figure 5.10. This section examines these features in detail.

The neurons in any of the NOT gates with a threshold of 0 (neuron C in Figure 5.4) are represented by wide black lines. These lines are only broken where an inhibitory signal arrives to the neurons in question; the neuron which fires the inhibitory signal can be seen directly above the black line and back one time-step (see Figure 5.11).

Many neurons in the SNNs presented in this chapter have thresholds of 1 and act as simple relays. The neurons in relay components (Section 5.2.1) and the input neurons of AND, OR, and NOT components (Sections 5.2.2, 5.2.3, 5.2.4, respectively) all behave in this way. Sometimes, a neuron acting as a relay is positioned in a spike plot directly below the neuron of a NOT component with a threshold of 0. This causes the relay neuron to fire continuously just like the neuron in the NOT component, except that it begins one or more time-steps later (see Figure 5.12). Also visible in Figure 5.12 is a gap in the relay neuron's row of the spike plot one time step after the NOT component's neuron stops firing.

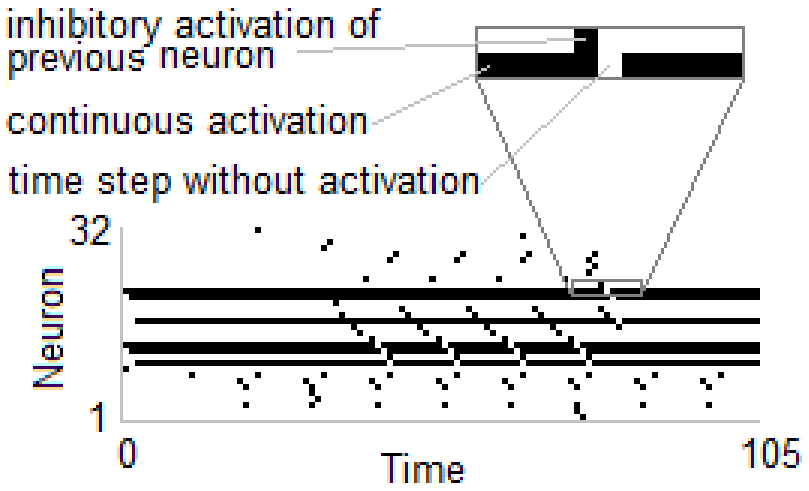


Figure 5.11: Detailed image of a continuously active neuron in a NOT component with a firing threshold of 0. The previous neuron sends an inhibitory signal to the continuously active neuron which suppresses it for one time step

Another type of feature visible in Figure 5.10 is the diagonal lines of firing neurons. These diagonal features even intersect the continuously firing neurons in NOT components as gaps in the plot of the NOT components' continuously firing neurons (see Figure 5.13). The diagonal features represent a chain of neurons firing one after another; the chain may cross component boundaries. The distance between one neuron firing in the chain and another neuron firing depends on the delay length of the synapse connecting them.

The most interesting type of feature shown in Figure 5.10 is when the diagonal lines mentioned above form a repeating pattern (see Figure 5.14). This typically indicates that a chain of connected neurons forms a cycle and behaves similar to the bistable memory loops mentioned in Section 3.2.9 of Chapter 3. Once one of these cyclical chains are initiated by an external pulse, the neurons within the cycle produce pulses with a fixed period.

5.6 Memory

The memory network (see Figure 5.15) was designed to be used as a component in a larger network which would in turn be used to solve the T-Maze problem (see Section 5.8). The memory network can be used to store an 8-bit value. It has 9 inputs: 1 for

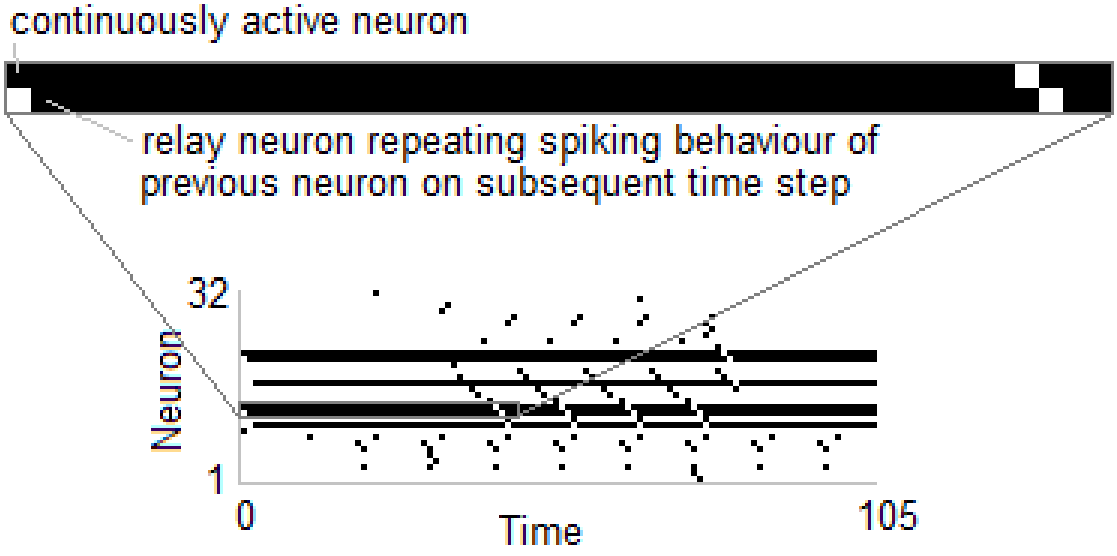


Figure 5.12: Detailed image of a neuron acting as a relay firing continuously as it receives a continuous signal from a NOT component

indicating that the memory should be set to a specific value and 8 representing the 8 values to set each bit to. It has 8 outputs, which continuously return the values being stored in the network.

The memory network was constructed by arranging eight flip-flop components in parallel; a global clock was added to the network to ensure that the internal clocks in the flip-flops were initiated. Although it has a simple design, it is the first network presented here which contains cycle-containing components (even if they are in an otherwise feed-forward network). Figure 5.16 shows that the memory network works as intended. Three sets of input signals (not including the initial pulse to CLOCK_IN) are injected into the network. The first set specifies that the circuit should store the number 202 (11001010). After the first input set is injected but before the second set is injected, the network outputs 202. The second input set specifies that the network should store the number 240 (11110000), and the network stores and outputs that number until the third set of inputs is injected. Finally, the network is instructed to store the number 0 (00000000) by the last set of inputs.

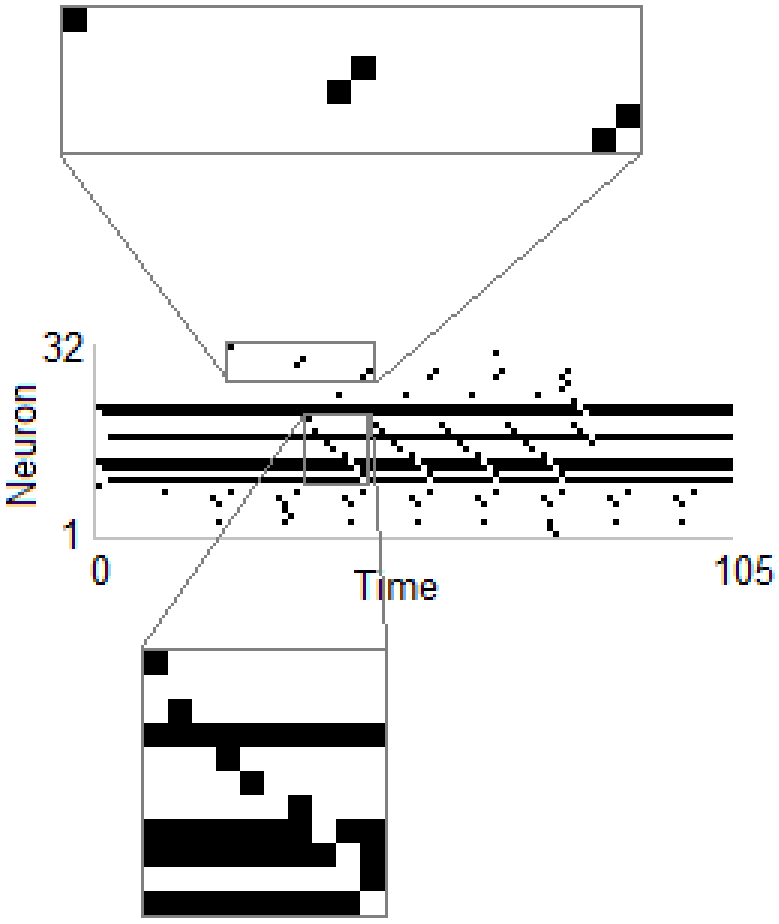


Figure 5.13: Image showing details of diagonal features representing chains of neurons firing one after another

5.7 Counters

Next, a series of networks were constructed which perform binary counting (similar to the counting task presented in (Ranhel et al., 2011)). These networks are noteworthy since they are implemented as networks containing both cycles and cycle-containing components. Here, all four axioms defined in Chapter 4 are being implemented successfully. In addition, the networks all perform as expected (Figure 5.21 shows the results of the most complex counter network), lending empirical support to Theorem 3.

Each of the counters here have global clocks added to them for simplicity's sake. A network's global clock initiates its components' internal clocks and also acts as an

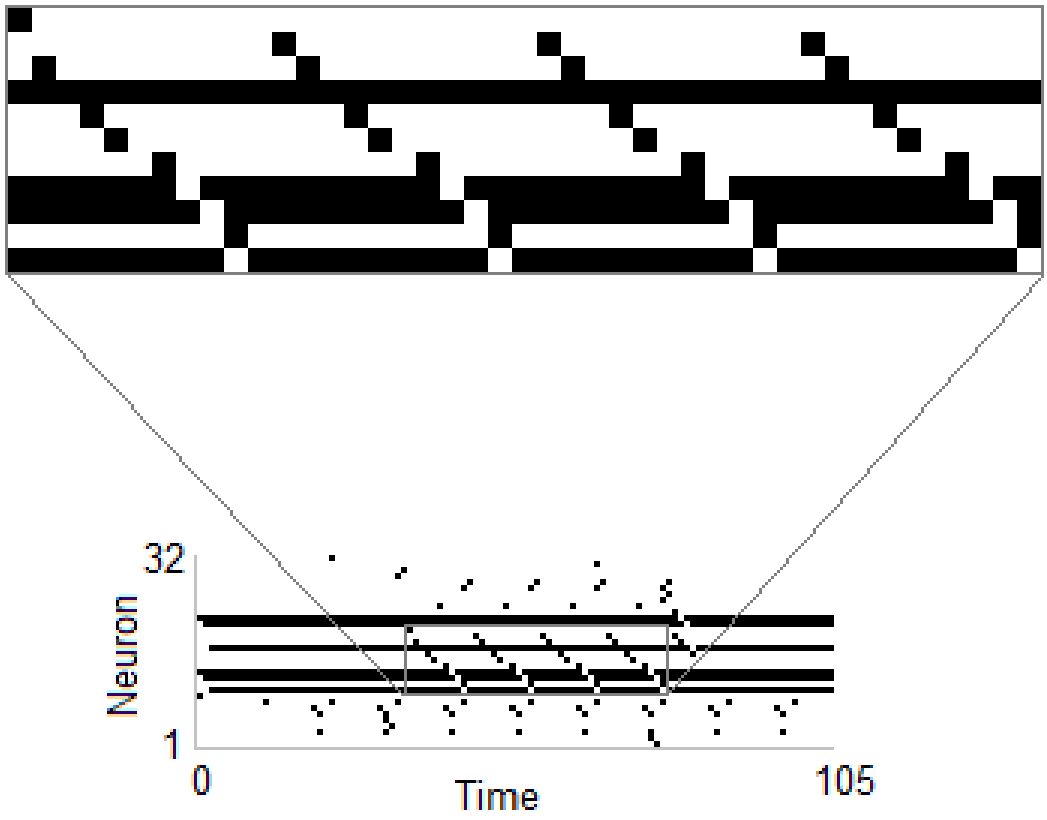


Figure 5.14: Image showing details of repeating diagonal features representing constructs similar to bistable memory loops

internal clock when the network is packaged as a component.

5.7.1 One Bit Counter

The simplest of the counter networks, the one-bit counter consists of a single flip-flop whose value is toggled every time a signal is introduced into the network through the INC (increment) input (see Figure 5.17).

Table 5.3 shows the constraints which were generated by the synfire circuit algorithm for the one-bit counter (before adding a global clock).

5.7.2 Two Bit Counter

The two-bit counter (see Figure 5.18) consists of a pair of flip-flops. Sending an increment signal into the network always toggles the first flip-flop's value (the least

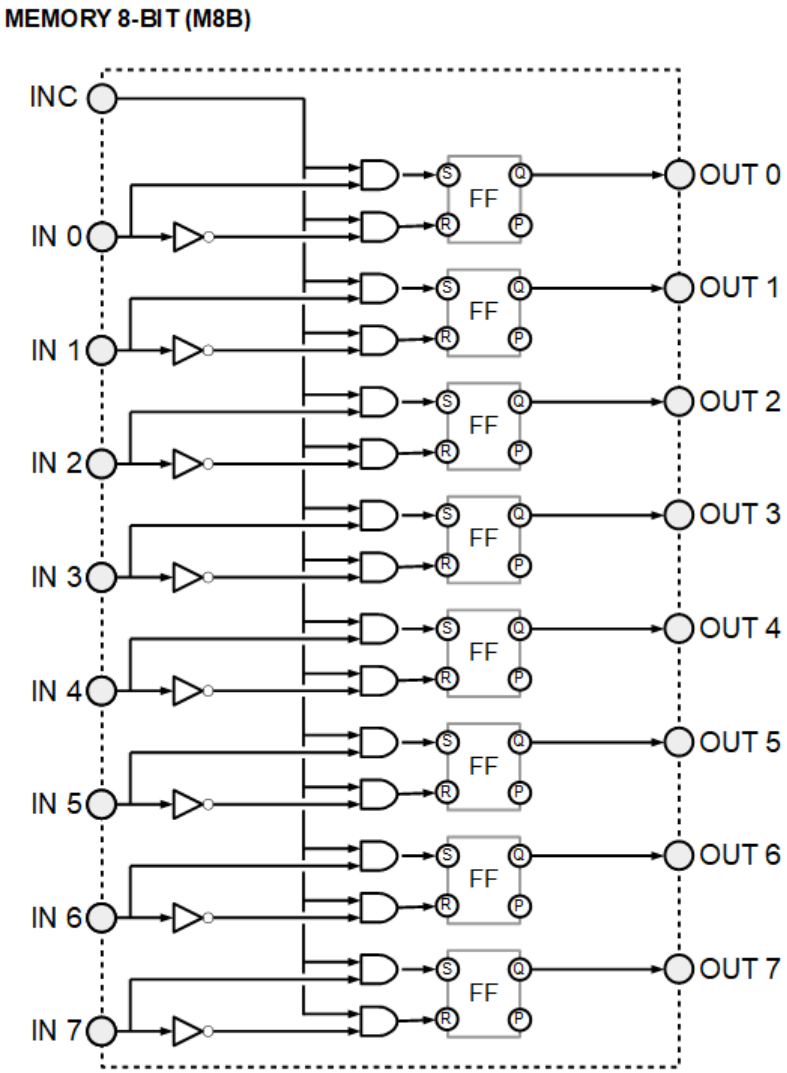


Figure 5.15: Spike plot of input and output neurons of the memory network

significant bit) and toggles the second flip-flop's value (the most significant bit) if and only if the first flip-flop is currently storing a value of 1.

5.7.3 Four Bit Counter

By combining two two-bit counters, a four-bit counter is constructed (see Figure 5.19). The increment signal is always sent to the first counter and sent to the second counter if and only if each bit in the first counter has a value of 1.

The four-bit counter shows that using the synfire circuit algorithm a group of nested, cycle-containing components can be combined in networks containing cycles;

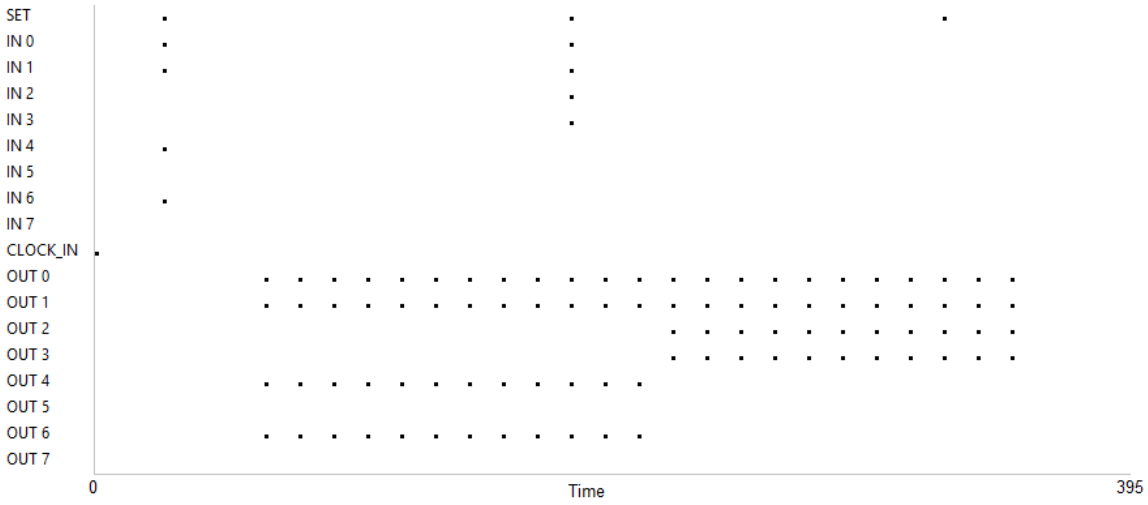


Figure 5.16: Spike plot of input and output neurons of the memory network

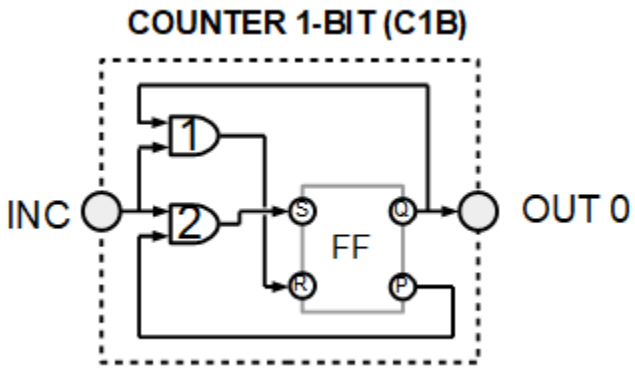


Figure 5.17: One-bit counter component diagram

the algorithm still implements the four axioms from Chapter 4 properly and the theorems from the same chapter continue to hold.

5.7.4 Eight Bit Counter

Combining two four-bit counters together (in the same manner that the two two-bit counters were combined to make a four-bit counter) makes an eight-bit counter (see Figure 5.20). The eight-bit counter was made to ensure that the synfire circuits continued to function with yet another layer of nesting.

Figure 5.21 shows a spike plot which includes the firing activity for input and output neurons in the eight-bit counter. The increment input always receives an input signal, and the output neurons collectively show a binary number which increases over

Path	Length of Path
AND_1 →C1B→AND_1	$V_{GlobalPeriod}$
AND_2 →C1B→AND_2	$V_{GlobalPeriod}$
C1B→AND_1 →C1B	$V_{GlobalPeriod}$
C1B→AND_2 →C1B	$V_{GlobalPeriod}$
INC→AND_1	V_1
FF→AND_1	V_1
INC→AND_2	V_2
FF→AND_2	V_2
AND_1 →FF	V_3
AND_2 →FF	V_3
FF→Out_0	$V_{GlobalPeriod}$

Component	Scale
FF	$V_{GlobalPeriod} \div$ FF.period

Table 5.3: Constraints for the one-bit counter circuit

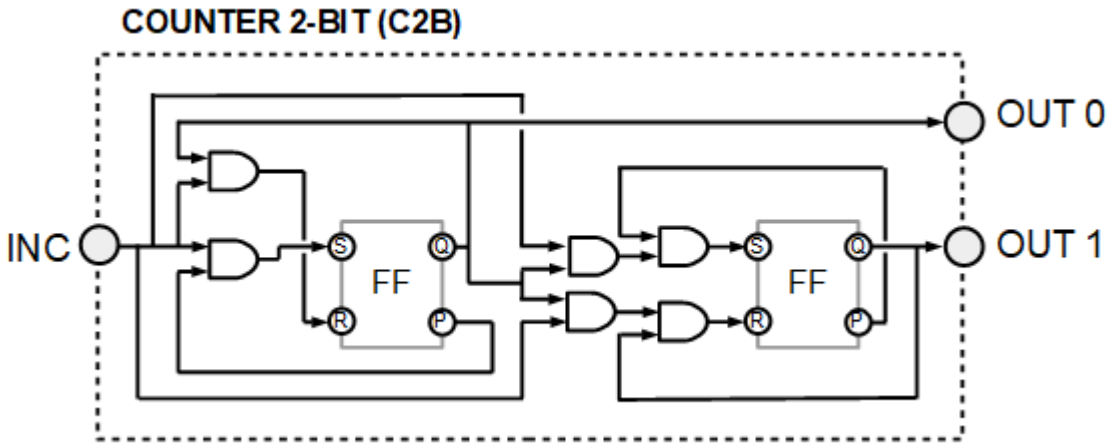


Figure 5.18: Two-bit counter component diagram

time. Note that time steps during which no output neurons fired are not shown to reduce the size of the plot.

5.8 T-Maze

Finally, the algorithms in Chapter 4 were used to construct a network which uses synfire circuits to solve the T-Maze problem. The T-Maze problem (Blodgett and McCutchan, 1947) is a problem where a mobile robot is placed in a maze shaped like a "T" and must navigate based on a cue it encounters in the environment. In the most simple version, the robot is placed at the base of the "T" and encounters a cue

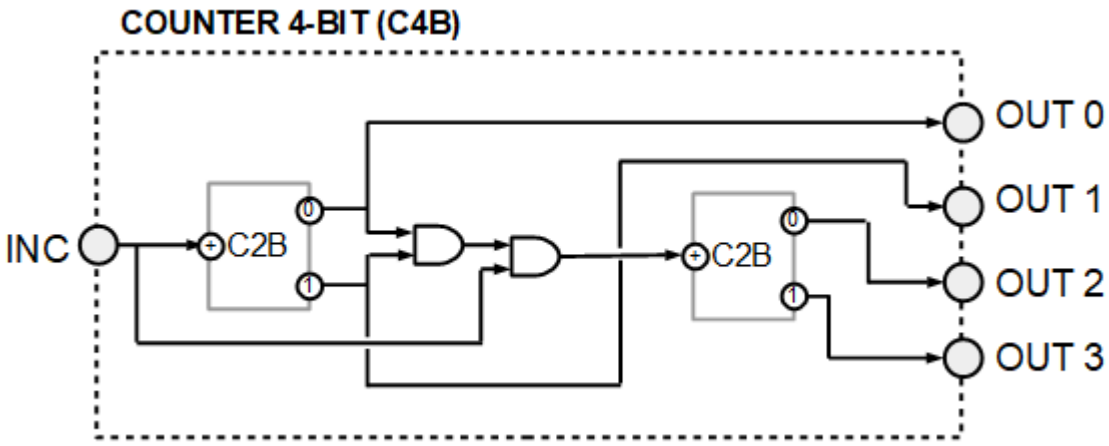


Figure 5.19: Four-bit counter component diagram

indicating whether it should turn left or right at the junction (see Figure 5.22). The robot is considered successful if it manages to turn in the correct direction and reach the desired endpoint. More general forms of the T-Maze problem can have more than one junction, reducing the likelihood of the robot successfully navigating the maze by chance (the likelihood of the robot succeeding by chance is equal to $\frac{1}{n}$, where n is the number of junctions in the maze).

In (Jeanson, 2013), Jeanson used the T-Maze problem to demonstrate how SNNs could be used as mobile robot controllers with dynamic memory capabilities. It was decided that the T-maze problem would be an a suitable test of the synfire circuit algorithm for a number of reasons. First and foremost, it would require designing and constructing a network to solve a specific problem; whereas some of the other networks examined in this chapter were constructed incrementally while developing the synfire circuit algorithm and therefore may not be the most reliable test of its functionality.

Specifically, the network described here which solves the T-Maze problem considers only the tasks of memory storage and retrieval. The network has one input which indicates that the cue has been located, nine inputs which contain the value of the cue, and a final input which indicates that the robot has reached a junction in the maze. These inputs are entered manually to the network as a bit-string, there is not actually a physical or simulated robot traversing a maze. Similarly, the network has two outputs: one indicating that the robot should turn left and the other indicating

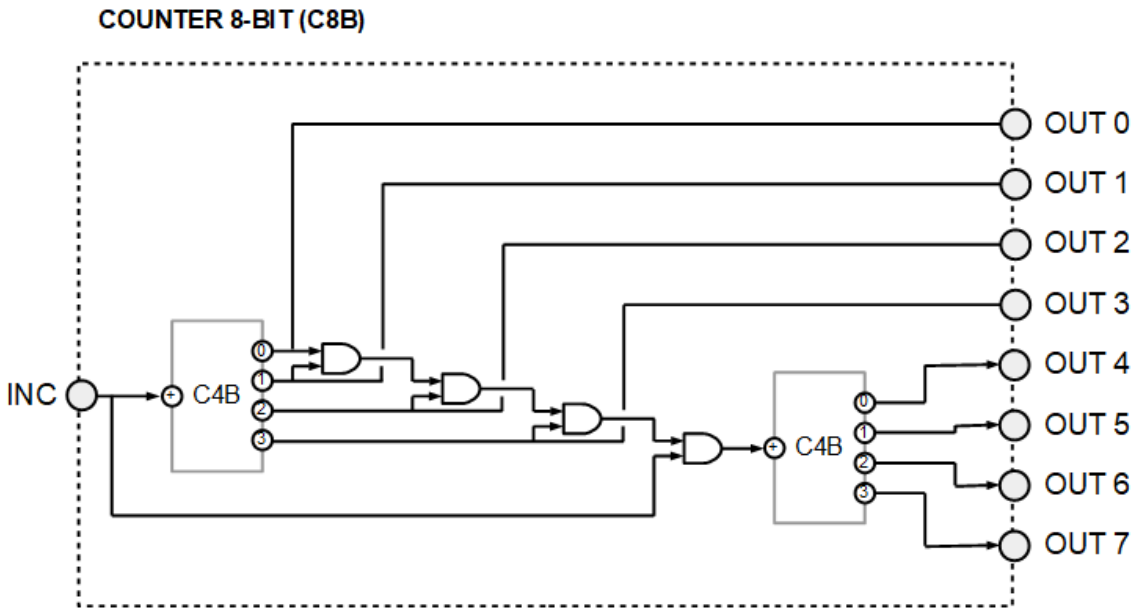


Figure 5.20: Eight-bit counter component diagram

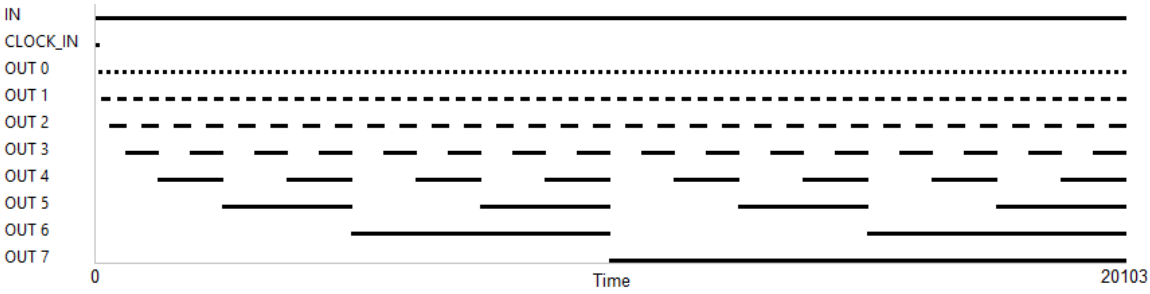


Figure 5.21: Spike plot for eight-bit counter network. Time steps during which no output neurons fired are not shown.

that the robot should turn right. Jeanson's research demonstrated the capability of SNNs to perform sensorimotor tasks (sensing the cue and manoeuvring the robot), so these tasks are omitted from the T-Maze problem in this thesis (if components were constructed to solve these tasks, then they could theoretically be added to the network with little difficulty).

5.8.1 Decoded Counter

The T-Maze solver is constructed from a number of components. The first of these is the decoded counter (see Figure 5.23). It is a three bit binary counter whose output

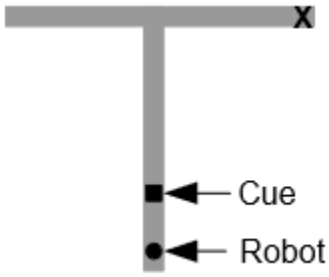


Figure 5.22: Setup for the T-Maze problem. The "X" represents the destination the robot is intended to reach

is decoded as a number stored in a one-hot array of bits (only one bit may store a 1; which bit is storing a 1 determines the value of the counter). This component is used to track the number of junctions which the robot has already navigated.

5.8.2 T-Maze Memory Unit

The decoded counter is used alongside a memory component (see Section 5.6) to form the memory unit for the T-maze solver (see Figure 5.24). The decoded counter is used to remember the number of junctions have been encountered, and the memory component stores the value of the cue (where each bit represents whether the robot should turn left or right at a given junction). The T-Maze memory unit has inputs for setting the memory component and incrementing the decoded counter component.

The T-maze memory unit performs a bit-wise AND operation on the outputs of the decoded counter and the memory components. This isolates the bit corresponding to the current junction. Next, OR operations are performed on the bits in the result of the bitwise AND. The result of the final OR operation is the output of the T-maze memory unit.

5.8.3 T-Maze Solver

The T-Maze solver (see Figure 5.25) is a network which simply passes inputs to a T-Maze memory unit component and interprets the output. If the output of the T-maze memory unit is 1 and the robot is detecting a junction, then the T-maze solver sends a signal to the "TURN LEFT" output. If the robot detects a junction and the T-maze memory unit outputs 0 then a signal is sent to the "TURN RIGHT" output.

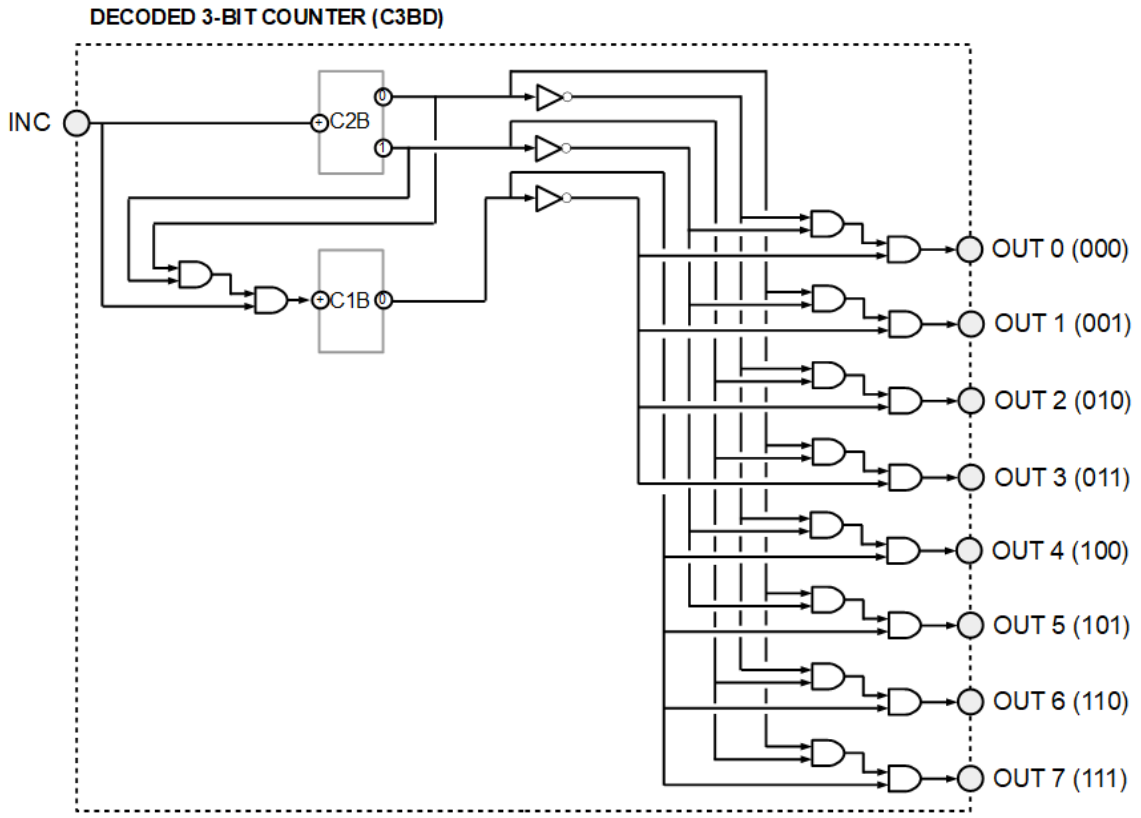


Figure 5.23: Decoded three-bit counter

If the robot is not at a junction then both outputs return 0 and the robot is meant to travel in a straight line.

One thing to note about this network is that there is a loop along the path from the "JUNCTION" input to the AND gate that checks the output of the T-Maze memory unit. The synapses in the loop have weights of 0 (represented by dotted lines). The loop does not actually compute anything but exists to force the constraint solver to make the path from the "JUNCTION" input to the AND gate take one global period longer. This was needed because the T-maze memory unit only outputs a signal once every two global periods. The extra global period of delay caused by the loop ensures that the signal from the "JUNCTION" input arrives simultaneously with the output from the T-maze memory unit. This does not mean that the synfire circuit algorithm does not work, since the signals from the "JUNCTION" input and the T-maze memory unit both arrive at the AND gate after an integer multiple of the global period (which is all the axioms and theorems, and by extension the algorithm,

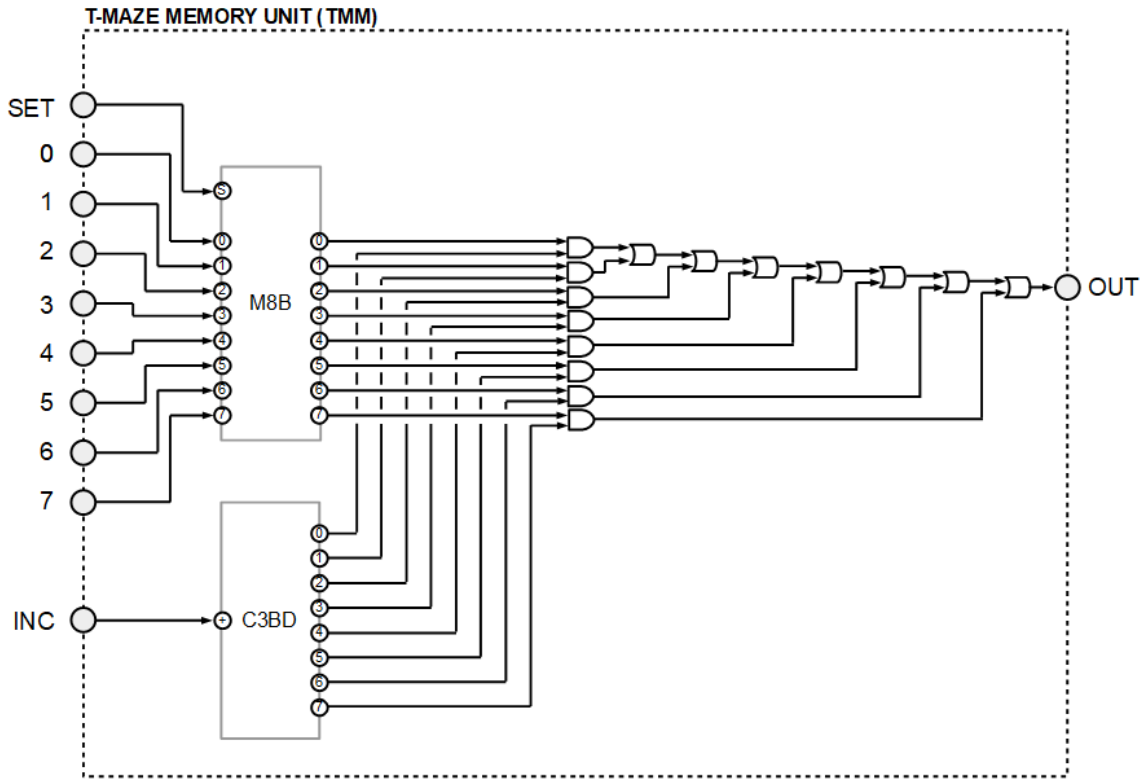


Figure 5.24: T-Maze memory unit

in Chapter 4 guarantee).

Figure 5.27 shows the layout of the maze to be solved. Figure 5.26 shows a spike plot for input and output neurons in the network used to solve the T-Maze problem. The first set of inputs (not including the pulse sent to the "CLOCK_IN" input) represents the robot encountering the cue. The cue provided indicates that the robot should take two left turns, two right turns, a left turn, a right turn, a left turn, and a final right turn (to reach the "X" shown in Figure 5.27). After, a series of pulses are injected into the "JUNCTION" input neuron; these represent the robot reaching a junction in the maze where it must decide which pathway to follow. The time between junctions represents the length of the hallway between the two junctions that the robot must traverse between the two junctions. Note that the hallways shown in Figure 5.27 are not drawn to scale. Hallways of different lengths were simulated to demonstrate that the T-Maze Solver reacts to junctions and does not simply output turn decisions after fixed time periods.

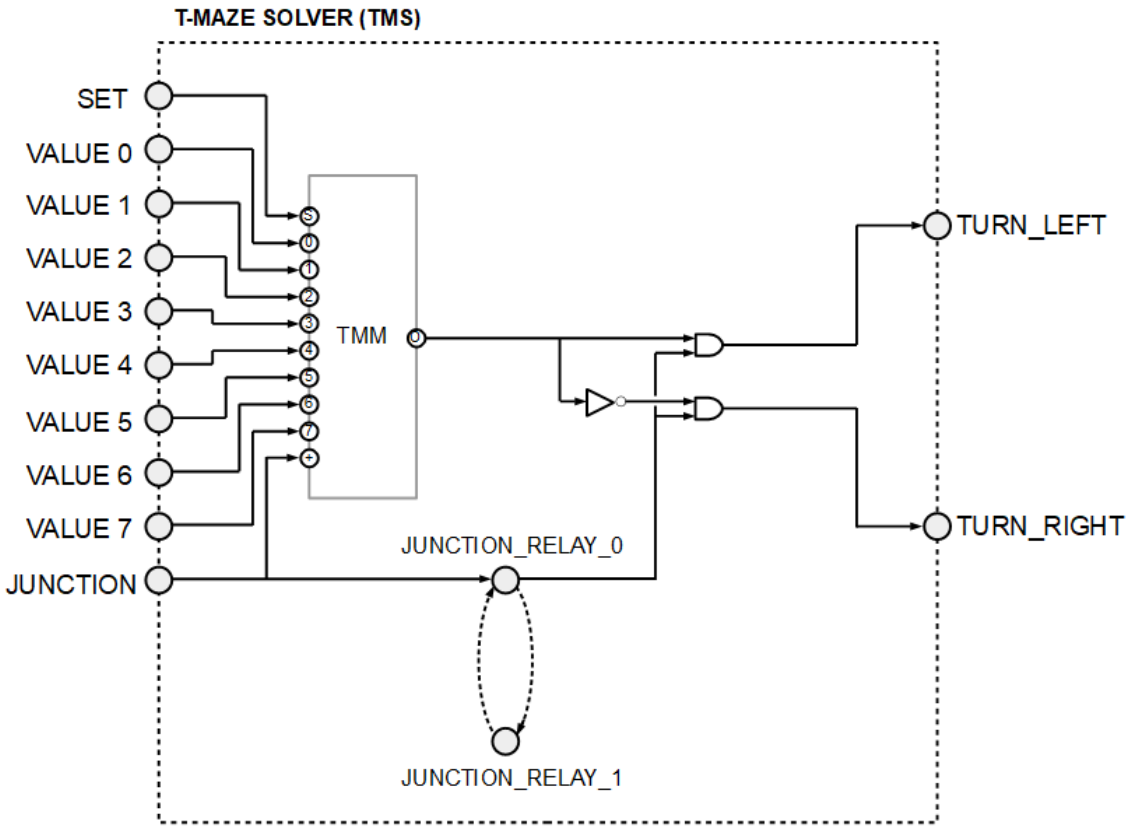


Figure 5.25: T-Maze solver

5.9 Applications

Although the research carried out in this thesis was of a theoretical nature and intended as a proof-of-concept, early results show that the synfire circuit algorithm can be used to produce networks which work well enough to solve practical problems such as the T-maze problem.

The experiments described in this chapter demonstrate the utility of the algorithm for combining component networks into larger, more complex networks. This can help the design and construction of complex networks, since if the sub-networks can be expressed as components then the desired behaviour will not have to be learned in a monolithic manner.

Additionally, it is theorised in Section 6.2 of Chapter 6 that this work could potentially be extended to help design modular neuronal assembly computing networks.

Finally, due to the utility of synfire chains, rings, and braids for analysing artificial

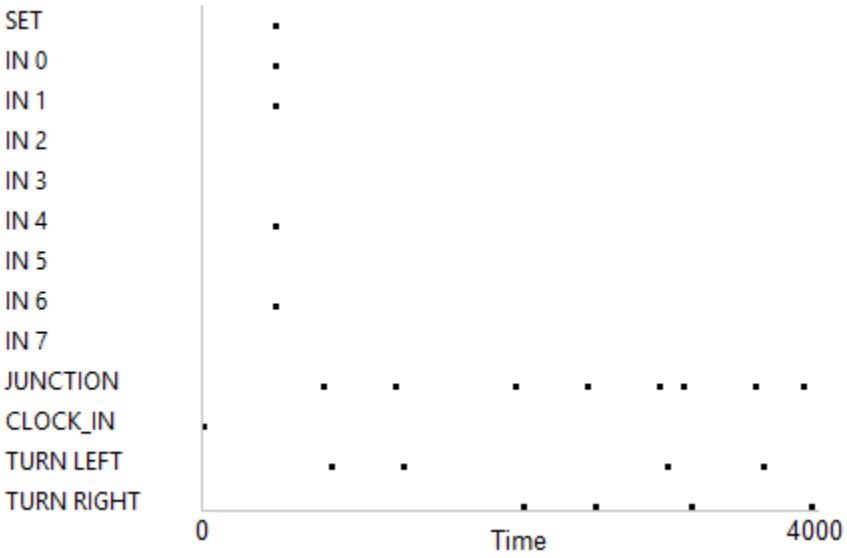


Figure 5.26: Spike plot showing spiking activity for input and output neurons in the network used to solve the T-Maze problem

and biological SNNs, it may be possible that synfire circuits could be used in a similar capacity.

5.10 Limitations

Although it appears to work quite well, the synfire circuit algorithm does have some limitations. These are discussed in this section.

First and foremost, the algorithm is limited by the type of neuronal groupings it can combine: components. The requirements for a grouping to be considered a component may seem somewhat draconian, but there may be a method to convert arbitrary groupings to components; exploring this possibility is left as future work and discussed in Section 6.2 of Chapter 6.

As mentioned in Section 5.8.3, a component may only output once per integer multiple of the global period. If paths leading from two such components firing on alternating iterations of the global period arrive at the same component, then the inputs will not arrive at the same time at the destination component. This can be mitigated easily enough by adding a "dummy" cycle as described in Section 5.8.3 to one of the paths once the problem is identified, but it would be ideal if the algorithm could account for this automatically.

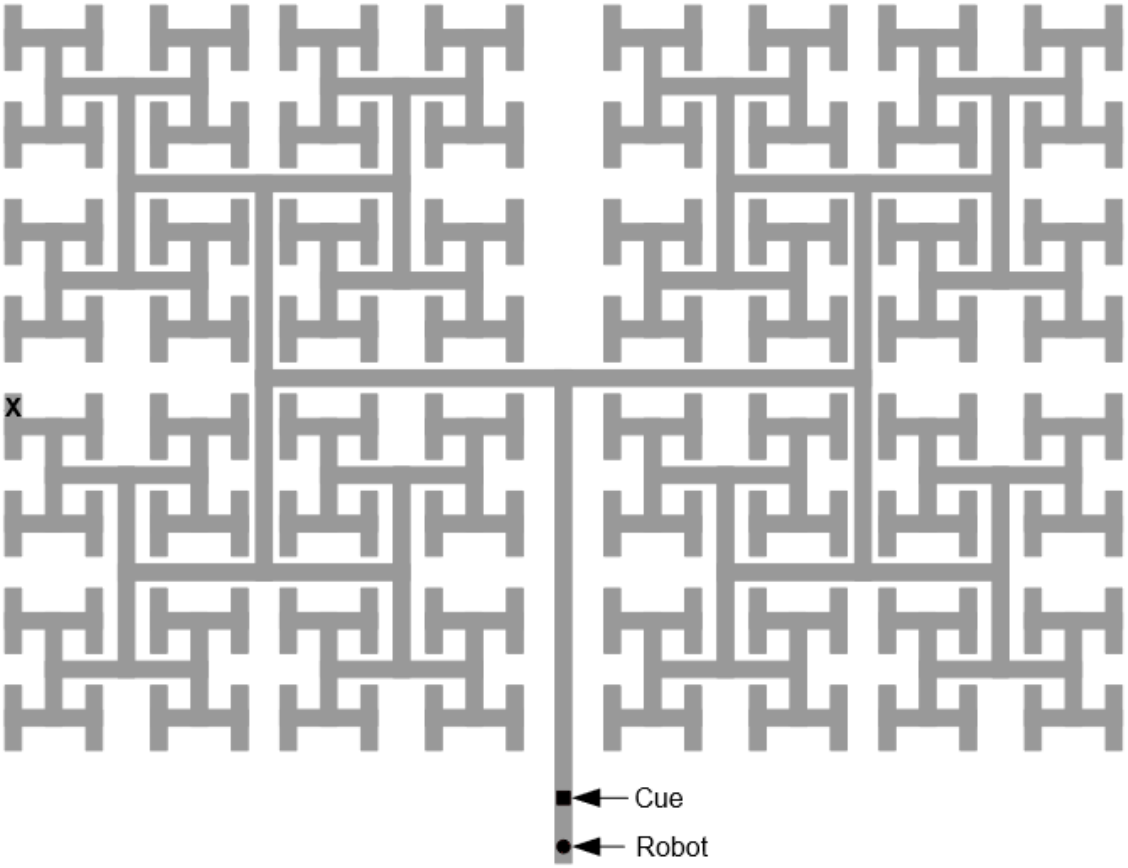


Figure 5.27: Setup for the variant T-Maze problem which is solved by the T-Maze Solver. The "X" represents the target that the robot must navigate to given the cue provided to the robot in the results shown in Figure 5.26.

The networks constructed to test the algorithm all perform digital logic operations, which require signals to be precisely timed. However, many applications of SNNs deal with noisy environments, where signals may not arrive precisely in sync with each other and the network as a whole. Since the synfire circuit algorithm assumes a neuronal model with a refractory period and leak time of 0, it is not robust to any kind of noise. This is a significant drawback, as one of the main features of SNNs is their capability to handle noise.

Chapter 6

Conclusion and Future Work

This thesis introduces a formal definition of a type of neuronal grouping called a component. Components are functional groupings of spiking neurons which exhibit specific properties and behaviours and were formally defined in Section 4.2 of Chapter 4. Components are useful as their properties allow them to be combined into composite SNNs using the synfire circuit algorithm.

Another result of this thesis is the synfire circuit, a generalization of synfire chains and braids to incorporate both inter-grouping and intra-grouping cycles; the groupings must take the form of components, however. Synfire circuits are described in Chapter 4, and partially satisfy the secondary objective of the research conducted in this thesis. Synfire circuits are useful as they can be used to help design complex SNN topologies from the bottom-up.

The primary objective of this thesis was to develop an automated technique for combining functional groupings of spiking neurons given the start and end points of inter-grouping synapses. This was partially accomplished. Rather than allowing entirely arbitrary neuronal groupings to be combined, the synfire circuit algorithm described in Section 4.5.1 of Chapter 4 allows the combination of components (described above). The tertiary objective of finding a technique to automatically specify inter-grouping synapse start and end points has not been completed.

Section 6.1 summarizes the results of the experiments carried out for this thesis. These results contribute empirical support for the correctness of the synfire circuit algorithm and related synfire circuit assembly model.

Section 6.2 provides an overview of work that could be done to build or improve on the research carried out in this thesis.

Java implementations of the algorithms and experiments described in this thesis can be found at:
https://github.com/adamebennett/dcdnn

6.1 Summary of Results

A number of experiments were performed to test the axioms and algorithms developed in this thesis. The experiments all consisted of combining two or more components into composite networks; initial experiments started by combining hand-made basic components and subsequent experiments built off of the initial experiments. The algorithm was shown to be capable of properly setting the delay lengths of inter-component synapses for a number of networks. The constructed networks included simple feed-forward networks which performed composite logic functions, a simple flip-flop capable of storing a binary value, various counters, and a network capable of solving a generalised variant of the T-Maze problem. These experiments provide empirical evidence that the axioms and algorithms are sound, and demonstrate the usefulness of the algorithm for designing SNNs for solving complex problems.

6.2 Future Work

The following sections discuss several potential avenues for expanding and improving on the research carried out in this thesis.

6.2.1 Converting Arbitrary Neuronal Groupings to Components

The most significant shortfall of the the synfire circuit algorithm is that it can only combine components. It would be ideal if it were capable of combining arbitrary neuronal groupings. There are several key properties that a component must have (see Chapter 4, Section 4.2). If a preprocessing technique could be developed to convert arbitrary groupings to components without changing the computation performed by the grouping, then this shortfall would be effectively negated. Some ideas are presented below to convert an arbitrary grouping to one which possesses the necessary attributes of a component:

Neurons, Synapses, Inputs, and Outputs

According to the definition of a component in Section 4.2 of Chapter 4, a component must have a set of neurons and a set of synapses. In addition, a subset of the neurons must be marked as input neurons and another subset must be marked as output

neurons. All of these properties are already present in functional neuronal groupings, so any conversion algorithm does not need to ensure that these properties are present.

Period

The period is potentially the most difficult property to assign to arbitrary neuronal groupings. There are two cases. In the first case, there are no cycles in the grouping and the period attribute can simply be set to -1.

If there are cycles in the grouping, then conversion becomes more complicated. If there are a limited number of cycles, then it may be feasible to scale (multiply the delay length of each synapse) by a constant such that the cycle with the smallest period has a period equal to an integer divisor of the periods of the other cycles.

Another possibility would be to look at the timing of spike pulses arriving at the output neurons of the grouping. This strategy involves plotting the output activity over a period of time while the grouping performs a range of operations. For the plot, consider each time step when any output neuron receives a signal as one where the grouping is firing. Then, take the plot and attempt to decompose the firing activity into a set of periodic signals that combine to form the overall plot (the overall plot need not fire at every interval of every component period, since components do not need to fire every period). Sparse coding may prove useful for such a task. Assuming that it is possible to decompose the firing activity into a number of component periods, it would then be possible to scale the component up until each period is an integer multiple of the smallest period; the scaled value of the smallest period could then be used as the period for the resulting component.

Delay

Delay is a fairly simple property to add to an arbitrary grouping. If the grouping contains no cycles, then the delay is simply the length of time it takes for a signal to travel from an input neuron to an output neuron. Note that the synapses leading to output neurons will need to be modified so that all possible routes through the grouping take the same length of time. This can be done by finding the longest route through the grouping and increasing the delay length of the last synapse of other routes such that the other routes are as long as the longest route.

If the grouping contains cycles, then the delay attribute of the resulting component is set to 0. However, in this case the delay lengths of synapses leading to output neurons will need to be increased such that all routes from either input neurons (if the route contains no cycles) or the final external input to the final cycle in the route (if the route contains a cycle) must have total delay lengths equal to the resulting component's period attribute.

In either case, constraint programming would likely prove to be a useful tool.

Periods_Per_Computation

If the component is to be used in a network which employs the algorithm for adding clocks specified in Section 4.6.1 of Chapter 4, then it will need to have a value for the *Periods_Per_Computation* attribute (otherwise a value of -1 should suffice). To calculate this value, simply compute the number of periodic sections (see Chapter 4, Section 4.5) in each cycle in the grouping and take the lowest common multiple of the results to be the *Periods_Per_Computation* value of the resulting component.

6.2.2 Automatically Assigning Start and End Neurons to Inter-Component Synapses

For the experiments conducted in this thesis, determining where inter-component synapses should start and end was fairly straightforward. However, some complex problems may not have readily apparent decompositions into components. In these cases, it would be ideal to have an algorithm for automatically designing hierarchical combinations of existing components so that the resulting network can perform some specified function.

To develop such an algorithm, it is first necessary to determine how to succinctly represent the behaviours of both components and networks. This would allow an algorithm to evaluate the behaviour of a combination of components against the desired behaviour of the overall network. Preliminary research suggests that finite state machines (FSMs) may be suitable for representing SNNs (both as components or composite networks); this is due to the fact that SNNs have inputs, outputs, and (sometimes) internal state.

Once low-level component behaviours and the high-level network behaviour have

been expressed as FSMs, a technique such as gene expression programming could be used to explore possible arrangements of components to perform the behaviour desired of the composite network.

6.2.3 Noise Tolerance

In this thesis, signals were always either inserted into the experimental networks with precise timing according to the networks' period or filtered out using global/internal clocks. This is, of course, not always feasible for practical networks where some amount of noise is to be expected. A substantial improvement to the synfire circuit algorithm would be to allow it to make networks capable of tolerating noise. One obvious approach would be to adapt the axioms in Section 4.5 of Chapter 4 and the synfire circuit algorithm to work with a neuronal model which includes leak-time and refractory period attributes.

Bibliography

Y Bengio and X Glorot. Understanding the difficulty of training deep feed forward neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256, 01 2010.

HC Blodgett and K McCutchan. Place versus response learning in the simple t-maze. *Journal of experimental psychology*, 37(5):412–422, 10 1947.

SM Bohte, JN Kok, and HL Poutré. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1):17 – 37, 2002. ISSN 0925-2312. doi: https://doi.org/10.1016/S0925-2312(01)00658-0. URL `http://www.sciencedirect.com/science/article/pii/S0925231201006580`.

RA Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

H Burgsteiner, M Kröll, A Leopold, and G Steinbauer. Movement prediction from real-world images using a liquid state machine. *Applied Intelligence*, 26(2):99–109, Apr 2007. ISSN 1573-7497. doi: 10.1007/s10489-006-0007-1. URL `https://doi.org/10.1007/s10489-006-0007-1`.

J Cabessa and P Masulli. Emulation of finite state automata with networks of synfire rings. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4641–4648, 2017. doi: 10.1109/IJCNN.2017.7966445.

T Caelli, L Guan, and W Wen. Modularity in neural computing. *Proceedings of the IEEE*, 87(9):1497–1518, Sep 1999. ISSN 0018-9219. doi: 10.1109/5.784227.

JPC Cajueiro and J Ranhel. Limits of coincidence detector neurons as decoders of polychronous neuronal groups firing completely. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–5, July 2015. doi: 10.1109/IJCNN.2015.7280572.

B Cessac, H Paugam-Moisy, and T Viéville. Overview of facts and issues about neural coding by spikes. *Journal of Physiology-Paris*, 104(1):5 – 18, 2010. ISSN 0928-4257. doi: https://doi.org/10.1016/j.jphysparis.2009.11.002. URL `http://www.sciencedirect.com/science/article/pii/S0928425709000849`. Computational Neuroscience, from Multiple Levels to Multi-level.

JK Chapin and MAL Nicolelis. Principal component analysis of neuronal ensemble activity reveals multidimensional somatosensory representations. *Journal of Neuroscience Methods*, 94(1):121 – 140, 1999. ISSN 0165-0270. doi: https://doi.org/10.1016/S0165-0270(99)00130-2. URL `http://www.sciencedirect.com/science/article/pii/S0165027099001302`.

J Chrol-Cannon, Y Jin, and A Grüning. An efficient method for online detection of polychronous patterns in spiking neural networks. *Neurocomputing*, 267:644 – 650, 2017. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2017.06.025. URL `http://www.sciencedirect.com/science/article/pii/S0925231217311530`.

P Diehl and M Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience*, 9:99, 2015. ISSN 1662-5188. doi: 10.3389/fncom.2015.00099. URL `https://www.frontiersin.org/article/10.3389/fncom.2015.00099`.

E Eskandari, A Ahmadi, and S Gomar. Effect of spike-timing-dependent plasticity on neural assembly computing. *Neurocomputing*, 191:107 – 116, 2016a. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2016.01.003. URL `http://www.sciencedirect.com/science/article/pii/S0925231216000503`.

E Eskandari, A Ahmadi, S Gomar, M Ahmadi, and M Saif. Evolving spiking neural networks of artificial creatures using genetic algorithm. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 411–418, July 2016b. doi: 10.1109/IJCNN.2016.7727228.

D Feldman. The spike-timing dependence of plasticity. *Neuron*, 75:556–571, 2012.

H Fujiia, H Itoa, K Aiharab, N Ichinoseb, and M Tsukadac. Dynamical cell assembly hypothesis — theoretical possibility of spatio-temporal coding in the cortex. *Neural Networks*, 9(8):1303 – 1350, 1996. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(96)00054-8. URL `http://www.sciencedirect.com/science/article/pii/S0893608096000548`. Four Major Hypotheses in Neuroscience.

M Garofalo, T Nieus, P Massobrio, and S Martinoia. Evaluation of the performance of information theory-based methods and cross-correlation to estimate the functional connectivity in cortical networks. *PLoS ONE*, 4:1–14, 2009.

GL Gerstein, ER Williams, M Diesmann, S Grün, and C Trengove. Detecting synfire chains in parallel spike data. *Journal of Neuroscience Methods*, 206:54–64, 2012.

JS Griffith. On the stability of brain-like structures. *Biophysical Journal*, 3:299–308, 1963.

Guo, X and Gao, X. A novel hierarchical ensemble classifier for protein fold recognition. *Protein Engineering, Design and Selection*, 21(11):659–664, 2008. doi: 10.1093/protein/gzn045.

LK Hansen and P Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, Oct 1990. ISSN 0162-8828. doi: 10.1109/34.58871.

G Hayon, M Abeles, and D Lehmann. A model for representing the dynamics of a system of synfire chains. *Journal of Computational Neuroscience*, 18:41–53, 2005.

DO Hebb. The organization of behavior, a neuropsychological theory, 1949.

Z Hu, T Wang, and X Hu. An stdp-based supervised learning algorithm for spiking neural networks. In D Liu, S Xie, Y Li, D Zhao, and ES El-Alfy, editors, *Neural Information Processing. ICONIP 2017. Lecture Notes in Computer Science*, volume 10635. Springer, Cham, 2017.

S Ito, ME Hansen, R Heiland, A Lumsdaine, AM Litke, and JM Beggs. Extending transfer entropy improves identification of effective connectivity in a spiking cortical network model. *PloS one*, 6:1–13, 11 2011.

EM Izhikevich. Polychronization: Computation with spikes. *Neural computation*, 18: 245–82, 03 2006.

F Jeanson. *Neural Coding via Transmission Delay Coincidence Detectors: An Embodied Approach*. PhD thesis, Carleton University, 2013.

N Kasabov, V Feigin, Z Hou, Y Chen, L Liang, R Krishnamurthi, M Othman, and P Parmar. Evolving spiking neural networks for personalised modelling, classification and prediction of spatio-temporal patterns with a case study on stroke. *Neurocomputing*, 134:269–279, 2014.

R Kompass. *Psychophysics Beyond Sensation: Laws and Invariants of Human Cognition*, chapter Universal Temporal Structures in Human Information Processing. Psychology Press, 2004.

N Kubota and H Sasaki. Genetic algorithm for a fuzzy spiking neural network of a mobile robot. In *2005 International Symposium on Computational Intelligence in Robotics and Automation*, pages 321–326, June 2005. doi: 10.1109/CIRA.2005.1554297.

JH Lee, T Delbruck, and M Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10:508, 2016.

G Ling, JP Anderson, and JP Sutton. A network of networks processing model for image regularization. *IEEE Transactions on Neural Networks*, 8(1):169–174, Jan 1997. ISSN 1045-9227. doi: 10.1109/72.554202.

D Liu and S Yue. Fast unsupervised learning for visual pattern recognition using spike timing dependent plasticity. *Neurocomputing*, 249:212 – 224, 2017. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2017.04.003. URL `http://www.sciencedirect.com/science/article/pii/S0925231217306276`.

B Lu and M Ito. Task decomposition and module combination based on class relations: a modular neural network for pattern classification. *IEEE Transactions on Neural Networks*, 10(5):1244–1256, Sep 1999. ISSN 1045-9227.

B Luitel and GK Venayagamoorthy. Cellular computational networks–a scalable architecture for learning the dynamics of large networked systems. *Neural networks : the official journal of the International Neural Network Society*, 50:120–123, 2014.

M Lukoševičius and H Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3:127–149, 2009.

Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997.

M Minsky, SA Papert, and L Bottou. *Perceptrons:An Introduction to Computational Geometry*. MIT Press, 2017.

NG Pavlidis, OK Tasoulis, VP Plagianakos, G Nikiforidis, and MN Vrahatis. Spiking neural network training using evolutionary algorithms. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 4, pages 2190–2194, July 2005. doi: 10.1109/IJCNN.2005.1556240.

S Peter, D Durstewitz, F Diego, and FA Hamprecht. Sparse convolutional coding for neuronal ensemble identification, 06 2016. URL `https://arxiv.org/abs/1606.07029`.

DA Pomerleau, J Gowdy, and CE Thorpe. Combining artificial neural networks and symbolic processing for autonomous robot guidance. *Engineering Applications of Artificial Intelligence*, 4:279–285, 1991.

F Ponulak. Resume - new supervised learning method for spiking neural networks. Technical report, Institute of Control and Information Engineering, Poznan University of Technology, 2005.

Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. URL `http://www.choco-solver.org`.

J Ranhel. Neural assembly computing. *IEEE Transactions on Neural Networks and Learning Systems*, 23(6):916–927, June 2012a. ISSN 2162-237X. doi: 10.1109/TNNLS.2012.2190421.

J Ranhel. *NEURAL ASSEMBLY COMPUTING and FINITE STATE MACHINES*, November 2012b.

J Ranhel. Neural assemblies and finite state automata. In *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence*, pages 28–33, Sept 2013. doi: 10.1109/BRICS-CCI-CBIC.2013.16.

João Ranhel, CV Lima, JLR Monteiro, JE Kogler, and ML Netto. Bistable memory and binary counters in spiking neural network. In *2011 IEEE Symposium on Foundations of Computational Intelligence*, pages 66–73, 04 2011. ISBN 9781424499809.

B Rekabdar, M Nicolescu, R Kelley, and M Nicolescu. An unsupervised approach to learning and early detection of spatio-temporal patterns using spiking neural networks. *Journal of Intelligent & Robotic Systems*, 80(1):83–97, Dec 2015. ISSN 1573-0409. doi: 10.1007/s10846-015-0179-1. URL `https://doi.org/10.1007/s10846-015-0179-1`.

DE Rumelhart, GE Hinton, and RJ Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

S Schliebs, A Mohemmed, and N Kasabov. Are probabilistic spiking neural networks suitable for reservoir computing? In *The 2011 International Joint Conference on Neural Networks*, pages 3156–3163, July 2011. doi: 10.1109/IJCNN.2011.6033639.

A Shrestha, K Ahmed, Y Wang, and Q Qiu. Stable spike-timing dependent plasticity rule for multilayer unsupervised and supervised learning. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1999–2006, 2017. doi: 10.1109/IJCNN.2017.7966096.

SB Shrestha and Q Song. Robust learning in spikeprop. *Neural Networks*, 86:54–68, 2017.

S Song, KD Miller, and LF Abbott. Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neuroscience*, 3:919–926, 2000.

Y Su, S Shan, X Chen, and W Gao. Hierarchical ensemble of global and local classifiers for face recognition. *IEEE Transactions on Image Processing*, 18:1885–1896, 2009.

H Sun, O Sourina, and G Huang. Learning polychronous neuronal groups using joint weight-delay spike-timing-dependent plasticity. *Neural Computation*, 28:2181–2212, 2014.

JP Sutton, JS Beis, and LEH Trainor. A hierarchical model of neocortical synaptic organization. *Math. Comput. Modeling*, 11:346–350, 1988.

A Taherkhani, A Belatreche, Y Li, and LP Maguire. Dl-resume: A delay learning-based remote supervised method for spiking neurons. *IEEE Transactions on Neural Networks and Learning Systems*, 26(12):3137–3149, Dec 2015. ISSN 2162-237X. doi: 10.1109/TNNLS.2015.2404938.

P Tino and AJS Mills. Learning beyond finite memory in recurrent networks of spiking neurons. *Neural computation*, 18(3):591–613, 2006.

C Trengove, M Diesmann, and C Leeuwen. Dynamic effective connectivity in cortically embedded systems of recurrently coupled synfire chains. *Journal of Computational Neuroscience*, 40(1):1–26, 2016. ISSN 1573-6873. doi: 10.1007/s10827-015-0581-5. URL `https://doi.org/10.1007/s10827-015-0581-5`.

G Valentini and F Masulli. Ensembles of learning machines. In *Proceedings of the 13th Italian Workshop on Neural Nets-Revised Papers*, volume 2486 of *WIRN VIETRI 2002*, pages 3–22. Springer-Verlag Berlin Heidelberg, 05 2002.

MC Vasu and EJ. Izquierdo. Evolution and analysis of embodied spiking neural networks reveals task-specific clusters of effective networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 75–82, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4920-8. doi: 10.1145/3071178.3071336. URL http://doi.acm.org/10.1145/3071178.3071336.

D Verstraeten, B Schrauwen, D Stroobandt, and J Van Campenhout. Isolated word recognition with the liquid state machine: a case study. *Information Processing Letters*, 95(6):521 – 528, 2005. ISSN 0020-0190. doi: https://doi.org/10.1016/j.ipl.2005.05.019. URL http://www.sciencedirect.com/science/article/pii/S0020019005001523. Applications of Spiking Neural Networks.

A Waibel. Modular construction of time-delay neural networks for speech recognition. *Neural Computation*, 1:39–46, 1989.

Q Wang and P Li. D-lsm: Deep liquid state machine with unsupervised recurrent reservoir tuning. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2652–2657, Dec 2016. doi: 10.1109/ICPR.2016.7900035.

P Zheng and J Triesch. Robust development of synfire chains from multiple plasticity mechanisms. *Frontiers in Computational Neuroscience*, 8:66, 2014. ISSN 1662-5188. doi: 10.3389/fncom.2014.00066. URL https://www.frontiersin.org/article/10.3389/fncom.2014.00066.

Z Zhou, J Wu, and W Tang. Ensembling neural networks: Many could be better than all. *Artificial Intelligence*, 137:239–263, 2002.