Real time Spaun on SpiNNaker

Functional brain simulation on a massively-parallel computer architecture

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

2016

Andrew Mundy
School of Computer Science

Contents

This thesis contains 29 904 **words.**

List of Figures

List of Tables

Abstract

Real time Spaun on SpiNNaker

Functional brain simulation on a massively-parallel computer architecture

Andrew Mundy

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY, 2016

Model building is a fundamental scientific tool. Increasingly there is interest in building neurally-implemented models of cognitive processes with the intention of modelling brains. However, simulation of such models can be prohibitively expensive in both the time and energy required. For example, Spaun – "the world's first functional brain model", comprising 2.5 million neurons – required 2.5 hours of computation for every second of simulation on a large compute cluster.

SpiNNaker is a massively parallel, low power architecture specifically designed for the simulation of large neural models in biological real time. Ideally, SpiNNaker could be used to facilitate rapid simulation of models such as Spaun. However the Neural Engineering Framework (NEF), with which Spaun is built, maps poorly to the architecture – to the extent that models such as Spaun would consume vast portions of SpiNNaker machines and still not run as fast as biology. This thesis investigates whether real time simulation of Spaun on SpiNNaker is at all possible.

Three techniques which facilitate such a simulation are presented. The first reduces the memory, compute and network loads consumed by the NEF. Consequently, it is demonstrated that only a twentieth of the cores are required to simulate a core component of the Spaun network than would otherwise have been needed. The second technique uses a small number of additional cores to significantly reduce the network traffic required to simulated this core component. As a result simulation in real time is shown to be feasible. The final technique is a novel logic minimisation algorithm which reduces the size of the routing tables which are used to direct information around the SpiNNaker machine. This last technique is necessary to allow the routing of models of the scale and complexity of Spaun. Together these provide the ability to simulate the Spaun model in biological real time – representing a speed-up of 9000 times over previously reported results – with room for much larger models on full-scale SpiNNaker machines.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy, in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations and in The University's policy on presentation of Theses.

Acknowledgements

I'm immensely grateful to all the friends, family and colleagues who have supported me over the past, very enjoyable, four years and, in particular, the last nine months while I've been located away from Manchester. I would have been utterly lost without the very generous open door policies, enthusiasm, guidance and oft needed criticism of my supervisors – Jim Garside, Simon Davidson and Steve Furber – and I owe much to the members of the APT group for their time and help – Steve Temple and Luis Plana especially.

Much of the work in this thesis is possible as a result of collaboration. Jonathan Heathcote and Jamie Knight have been great friends, and wonderful collaborators, who I will miss sharing an office with. Terry Stewart, Chris Eliasmith and others at the University of Waterloo have been immensely kind and I look forward to joining them in due course. It's been a pleasure to work with Jörg Conradt and others.

This work would not have been possible if not for the Engineering and Physical Sciences Research Council, the School of Computer Science and various other bodies who funded my studies and travel.

Finally, I'd like to thank my parents for their proof-reading, support and love, and my wonderful wife Beccy for sharing the first four years of our marriage with this thesis.

Chapter 1

Introduction

Building and simulating models of the brain is a promising way to investigate the principles which underlie human cognition. Although this is not a recent observation (Turing 1950) developments in computing technology have made simulation of larger and more detailed models increasingly tractable. These technological advances have catalysed computational neuroscience and resulted in the formation of both the European Human Brain Project (Markram et al. 2011) and the US BRAIN Initiative (The White House 2013) – two 'mega projects' dedicated to significantly improving understanding of the brain through simulation of computer models. These models can be broadly classified as those which build "up" from a description of neural form (e.g., Izhikevich and Edelman 2008) and those which focus on function and build "down" to tie *behaviour* to a *neurally-inspired* implementation (e.g., Verschure and Voegtlin 1998; Sandamirskaya 2013).

The leading model in this latter field is, arguably, Spaun (Eliasmith, Stewart, et al. 2012; Eliasmith 2013), which, unlike many *cognitive* models, draws together ideas from classical Artificial Intelligence (e.g., Fodor and Pylyshyn 1988; Anderson et al. 2004), Artificial and Spiking Neural Networks (e.g., McCulloch and Pitts 1943; Rosenblatt 1958; Hopfield 1982; Maass 1997) and dynamic systems theory (Gelder 1998). Consisting of 2.5 million neurons and a virtual 'eye' and 'arm', Spaun is capable of performing eight tasks, in which it is apparent that it reproduces some characteristics of human performance. For example, humans have been shown to be more likely to forget the middle, rather than the beginning or the end, of lists they have remembered (Dosher 1999). Spaun shows a similar tendency, implying that it captures something about how the human brain memorises lists. To truly assess this requires many more trials than have been performed.

Alongside these developments in the fields of computing and cognitive modelling has been a growing recognition that *behaviour* is inextricably tied to *environment* (Braitenberg 1986; Brooks 1990; Webb 2001; Kaplan 2008). While the environments in which these neural models are *embodied* could be entirely virtual, the growing range of biologically-inspired (*biomimetic* and *neuromorphic*) sensors (e.g., Chan, Liu, and Schaik 2007; Lichtsteiner, Posch, and Delbruck 2008) suggests that the environment could be our environment and the neural models used to control robotic agents. For example, the virtual eye and arm to which Spaun is connected (Figure 1.1) could be replaced with a biomimetic retina and a robotic arm. For this to be realistic simulations of neural models that run as fast as biology will be required.

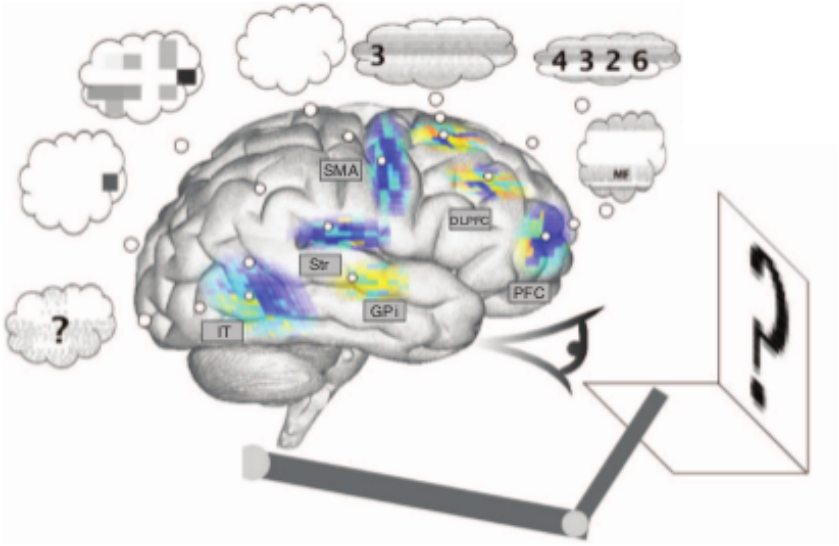


Figure 1.1 – The Spaun model, reproduced from Eliasmith, Stewart, et al. (2012).

Unfortunately, complex neural networks are rarely simulated in biological real time, meaning that the time taken to perform the computation associated with the simulation is often significantly greater than the 'time' that has passed in the model. For example, the Spaun model required 2.5 hours of compute for every 1 s of the simulation (in 2012, as reported by Stewart and Eliasmith 2014) – i.e., 9000 times slower than biology. Improvements are possible, indeed an improvement of an order of magnitude has been informally reported, but the technologies involved suffer from severe drawbacks. Supercomputers and General Purpose Graphics Processing Units (GPGPUs) consume much power and have limited scalability. In contrast, hardware implementations of neurons (Choudhary et al. 2012; Merolla et al. 2014; Meier 2015) consume little power but are too inflexible

to allow new types of neural model to be investigated. Since Spaun is less than 1 % of the size of the human brain, the abilities of simulation technologies to scale and adapt is crucial if ever-larger and more complex models are to be constructed and simulated.

SpiNNaker (Furber, Galluppi, et al. 2014) – a low power, massively parallel computer designed for the real time simulation of large scale spiking neural networks – may prove to be a solution. A SpiNNaker machine consists of up to a million highly constrained processing cores connected to a customised network. This network is optimised, using a 'multicast' architecture, for the efficient simulation of the high degree of neural interconnectivity found in brains. While SpiNNaker has proven to be suitable for the simulation of a wide range of neural networks (Sharp, Petersen, and Furber 2014; Stromatias et al. 2015; Knight, Tully, et al. 2016) the architecture is not well suited to the simulation of networks like Spaun (Section 4.1).

Construction of Spaun was guided by two theories: the Neural Engineering Framework (Eliasmith and Anderson 2004) and the Semantic Pointer Architecture (Eliasmith 2013). The Neural Engineering Framework is a series of principles which describe how a functional description of a system may be implemented in a network of spiking neurons (Section 2.2) and the Semantic Pointer Architecture describes how high dimensional vectors can be used to represent and manipulate *symbol*-like values (Section 2.3). Unfortunately, networks constructed using these principles act to stress the SpiNNaker architecture. In particular, the dense neural interconnectivity and high firing rates they require consume much more memory, network bandwidth and compute time than would be needed for the types of neural networks for which SpiNNaker was designed (Sections 3.3 and 4.1). To manage these loads a choice must be made between simulating the network slower than biology, acquiring a larger SpiNNaker machine, limiting the scale of the neural models which can be investigated or ignoring neurons altogether and simulating merely the "computational" elements of Spaun. Unfortunately, each of these choices presents drawbacks: simulating the network slower than biology precludes interaction with biomimetic sensors and actuators; acquiring (or building) a larger computing system may be prohibitively expensive; limiting the scale of the neural network limits the scale and complexity of system which may be examined; replacing simulation of neurons with simulation of "computational" elements alone can change the *behaviour* of the system and reduce the explanatory strength of the model and results (Eliasmith, Stewart, et al. 2012, p. 219; Stewart 2012).

This thesis presents techniques which reduce the memory, compute and network loads required to simulate neural networks built with the Neural Engineering Framework and Semantic Pointer Architecture. Specifically, it demonstrates that a core component of the Spaun network can be simulated *in biological real time* (meaning that 1 s of simulation requires 1 s of computation) using one twentieth of the resources that would have been required had SpiNNaker been used as intended. As a consequence, SpiNNaker is shown to be an ideal platform for future research on Spaun and its successor models. In particular, since the scale of machine expected to be necessary to simulate Spaun is small enough, and consumes sufficiently little power, it may be situated in the same room as a robot, allowing Spaun (or similar) models to interact with an environment through a tight control loop. Alternatively, many instances of the Spaun model could be run in parallel, allowing more data to be gathered about its performance and enabling more meaningful comparisons with results from biology and psychology. Finally, there is scope to increase the scale or complexity of Spaun allowing it to perform more tasks and again increasing potential comparisons with the empirical sciences.

These reductions in the memory, compute and networks loads consumed by neural simulation are made by exploiting a characteristic of the Neural Engineering Framework to change the way in which simulation data is transferred amongst processing cores (Chapter 4). As a result, the architectural target of 1000 neurons per core is more than doubled and the memory required to represent a neural network significantly reduced (Mundy, Knight, et al. 2015). This increases how densely neural models may be packed into a SpiNNaker machine, allowing larger models to be simulated using fewer processors.

Unfortunately, the improvements made by changing the way in which processors communicate are still not enough to allow real time simulation of neural models like Spaun (Chapter 5). This is because the dense interconnectivity of models built with the Semantic Pointer Architecture requires much more network bandwidth and processing than are available. Managing these loads requires that the simulation run slower than biology. However, by replacing the dense network connectivity with an 'interposer' the traffic and compute loads can be reduced sufficiently to allow simulation in biological real time. The interposer proposed in this thesis exploits the changed values which are communicated by processors and the *multicast* capability of the SpiNNaker network to distribute simulation state over a wide area. This reduces both the overall traffic and the

fan-out and fan-in of individual processors, which in turn reduces local traffic and the processing required.

Despite these improvements, the interposer still requires some degree of traffic fan-out and fan-in and this results in the *routing tables* which are used to direct traffic around the SpiNNaker network becoming too large to store on the chip (Chapter 6). Although logic minimisation can be used to reduce the size of these routing tables (Liu 2002) existing techniques result in insufficient minimisation of benchmark SpiNNaker routing tables. This thesis introduces a new technique which, by exploiting the ordered nature of the tables, achieves a greater degree of minimisation (Section 6.3). As serial application of this technique to all the tables in a million core machine would be prohibitively expensive, the massive parallelism of SpiNNaker can be exploited to perform this task. As the limited memory available to a SpiNNaker core precludes the use of any of the existing minimisers this thesis introduces a new algorithm. Parallel use of this algorithm on SpiNNaker is shown to minimise one of the benchmark sets of routing tables 64 times faster than a desktop PC (Mundy, Heathcote, and Garside 2016).

The work presented in this thesis builds towards biological real time simulation of "the world's first functional brain model" (Stewart and Eliasmith 2014), Spaun (Eliasmith, Stewart, et al. 2012), on SpiNNaker. This is a major milestone, not only for the SpiNNaker project, but also, since it paves the way for simulation of models *larger and more complex* than Spaun, in our quest to understand cognition.

Contributions

Mundy, Andrew, James Knight, Terrence C. Stewart, and Steve Furber (2015). "An efficient SpiNNaker implementation of the Neural Engineering Framework". In: *Neural Networks (IJCNN), 2015 International Joint Conference on*. DOI: 10.1109/IJCNN.2015.7280390. – Nominated for Best Paper Award

Mundy, Andrew, Jonathan Heathcote, and Jim D. Garside (2016). "On-Chip Order-Exploiting Routing Table Minimization for a Multicast Supercomputer Network". In: *High Performance Switching and Routing (HPSR), 17th International Conference on*. – **Best Paper Award**

Stewart, Terrence C., Ashley Kleinhans, Andrew Mundy, and Jorg Conradt (2016). "Serendipitous Offline Learning in a Neuromorphic Robot". In: *Frontiers in Neurorobotics* 10 (1). ISSN: 1662-5218. DOI: 10.3389/fnbot.2016.00001.

Knight, James, Aaron R. Voelker, Andrew Mundy, and Chris Eliasmith (2016). "Efficient SpiNNaker simulation of a heteroassociative memory using the Neural Engineering Framework". In: *Neural Networks (IJCNN), 2016 International Joint Conference on*.

Further publications based on the contents of this thesis are planned.

In addition:

- An implementation of the algorithm presented in Chapter 6 has already been used by other researchers in Manchester to minimise the routing tables arising from a detailed model of a part of the human brain (Potjans and Diesmann 2012).
- Various insights arising from this work are being fed into design of the next generation of the SpiNNaker hardware and support software built – in collaboration with others – for this project is used by others in the group (Knight and Furber 2016).
- The implementation of the Neural Engineering Framework described in Chapters 4 and 5 is in active use by a number of researchers across the globe.

Chapter 2

Neurons, synapses and representation

2.1 Modelling the nervous system

The human brain is thought to contain of the order of 85 billion neurons (Herculano-Houzel 2009). Each neuron is a complex electrochemical system capable of altering the ratio of ions and hence the electric potential across its cellular membrane. From the main cell body of a neuron grows a tree-like structure of inputs, named *dendrites*, in addition to which each neuron possesses a single (often long) output structure, the *axon*. Varying electric potentials on the dendrites will modify the potential of the cell itself, eventually causing a *spike* or series of spikes of potential to be generated and pass down the axon, affecting further neurons via *synapses* – the connections between neurons.

In computational modelling of the nervous system it is common to use a system of coupled differential equations to describe a single neuron. One of the earliest models was proposed by Hodgkin and Huxley (1952). However, this model is computationally expensive and while simplifications exist there are wide variations in the types and characteristics of neurons found within different regions of the brain and accounting for this range of variation within a single model is challenging. Moreover, there is debate as to exactly which features of neurons are relevant for computational models (Izhikevich 2004). Since there is controversy about which features of the neural behaviour are relevant the *flexibility* resulting from software simulation of the nervous system may well outweigh the speed and power benefits of dedicated hardware – see Chapter 3.

Leaky Integrate-and-Fire model

The Leaky Integrate-and-Fire (LIF) model of the neuron (Lapicque 1907; Abbott 1999) is a simple neuron model which captures some of the key features of the behaviour of biological neurons and is a good illustration of how a typical neuron works. LIF neurons will be used in all models in this thesis since it was the only model used in Spaun (Eliasmith, Stewart, et al. 2012). The techniques described in this thesis are, however, amenable for use with other neural models. Likewise, it is possible to construct Spaun from alternative neuron models (Eliasmith, Gosmann, and Choo 2016).

The LIF neuron can be thought of as the combination of a first-order low-pass filter and a 'spike generator'. Input to the neuron is filtered using the low-pass filter. Should the output of the filter pass a given *threshold* value the neuron emits a *spike* and the output of the filter is reset to a lower value and held there for a time. We will focus on a *normalised* LIF model in which the output of the filter, representing the cell membrane voltage, is allowed to vary between zero and one. As a consequence of this the 'currents' and 'voltages' discussed below are analogues of the real quantities of the LIF neuron and are unitless.

Figure 2.1 shows a simulation of a LIF neuron fed by a constant current, rather than by the spikes of other neurons. As can be seen, the input value (uppermost panel) is filtered to form the membrane 'voltage' (middle panel) and once the 'voltage' passes the threshold a *spike* is emitted (lowest panel) and the voltage is reset to zero for a period. The time constant of the low-pass filter is τ_{rc} and the duration for which the voltage is held at zero after a spike is called the *refractory period*, τ_{ref}.

While the neuron is neither firing nor in the refractory period the 'membrane voltage' (in this case a dimensionless quantity) is described by the equation:

$$\dot{v} = \frac{i_{\text{input}} - v}{\tau_{\text{rc}}} \tag{2.1}$$

where i_{input} is the input to the neuron (and is a dimensionless quantity analogous to the 'current' in the non-normalised LIF model). This input is a combination of a bias, i_{bias}, and the spiking-output of other neurons. However, for our purposes it is useful to express this as $i_{\text{input}} = gx + i_{\text{bias}}$ where g is called the *gain* of the neuron and x is a value over which we have control.

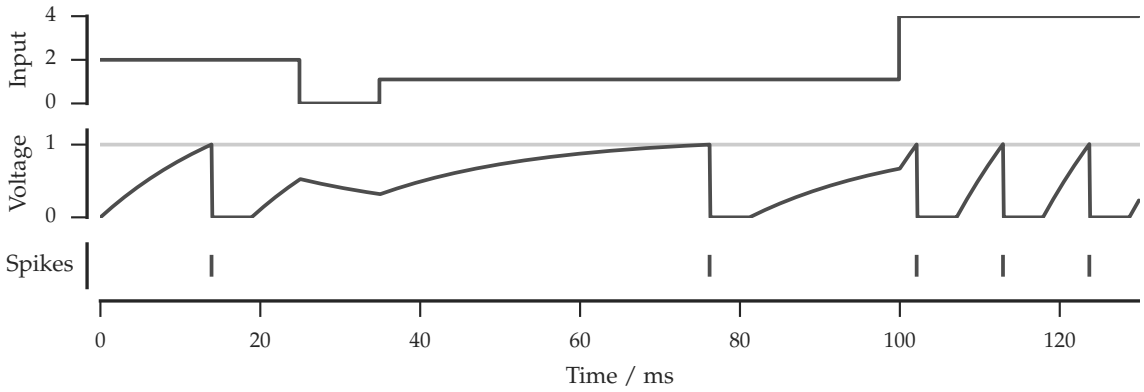


Figure 2.1 – Simulation of a Leaky Integrate and Fire (LIF) neuron. The top panel shows the input to the neuron which, for illustrative purposes, is a piecewise function rather than the accumulated spikes of other neurons. The middle panel shows the membrane 'voltage' – formed as a 'leaky' integral of the input with time constant τ_{rc}. Once the voltage passes a threshold a *spike* is emitted (bottom panel) and the membrane voltage held low for a period of time called the *refractory period*, τ_{ref}. (For this illustration $\tau_{\text{ref}} = 5\,\text{ms}$ and $\tau_{\text{rc}} = 20\,\text{ms}$).

Tuning curves

One of the characteristics of the LIF model is that when fed a constant input it produces spikes at a constant rate (see the final 30 ms of Figure 2.1). This allows us to draw a graph describing the relationship between constant input on the x-axis and firing rate (in Hz) on the y-axis. The line represented on this graph is called the *tuning curve* of the neuron.

Figure 2.2 shows a number of tuning curves which illustrate the effect that varying i_{bias} and g has on the characteristic response of the neuron. Through modification of the bias current, i_{bias}, the value of x for which the neuron starts to fire can be selected (Figure 2.2(a)). By modifying the gain of the neuron, g, the slope of the tuning curve can be made steeper or shallower (Figure 2.2(b)). By varying both the gain and the bias a range of neural responses can be generated.

It is important to note that while the neuron response is characterised in terms of spike rates this is merely an analytic convenience and all simulations in this thesis are run using spiking models of neurons rather than "rate" neuron models.

Neural networks and synapses

When a biological neuron *fires* the pulse that is generated travels along the axon of the neuron. To make contact with the dendrites of other neurons the axon is branched, with each branch leading to junctions with other neurons. Each junction between the axon of

the *presynaptic* neuron and the dendrites of the *postsynaptic* neuron is called a *synapse*. At a synapse the spike generated by the neuron is chemically transmitted across a short gap and the transmission method is capable of varying both the strength and the shape of the impulse that is received by the postsynaptic neuron.

Instead of modelling synapses directly it is common to specify them in two ways: The change in strength of the impulse caused by a synapse is called the synaptic *weight*; the pulse-shaping is often referred to as *synaptic filtering* and can be modelled as a linear transfer function. Note that the transfer function used to model synaptic shaping is distinct from the transfer function used to model the changes in the membrane voltage of the LIF neuron. If multiple spikes travel across synapses to the same neuron (the *postsynaptic* neuron) their contributions to the input of the neuron are summed. Figure 2.3 shows the shaping, weighting and summing applied to three spikes arriving at the same postsynaptic neuron. In this example the synaptic shaping was performed with a second-order filter; the Spaun model, however, uses first-order filters and these are used throughout the remainder of this thesis.

Neurons rarely occur in isolation; it is common to think of them forming groups, or *populations*, which work together to fulfil some function. The synaptic weights between two populations of neurons can be described by a *synaptic weight matrix*, one of which is illustrated in Figure 2.4 in which a black dot indicates a synaptic connection between two neurons.

2.2 The Neural Engineering Framework

The base components of the nervous system are, of course, only part of the story about behaviour and cognition. There is a wealth of studies which indicate the utility of even small nervous systems in behavioural responses (e.g., Kandel 1976), but tying more complex behaviours to their neural implementation has proven much more complex. The Neural Engineering Framework (NEF) (Eliasmith and Anderson 2004; Stewart and Eliasmith 2014) is one attempt.

The Neural Engineering Framework provides a toolkit that allows the conversion of a functional description of a *system* into a neural implementation or *model*. Such a model consists of populations of neurons (or *ensembles*) which act to represent the *variables* of the system. These ensembles produce spikes which are transmitted to other ensembles by

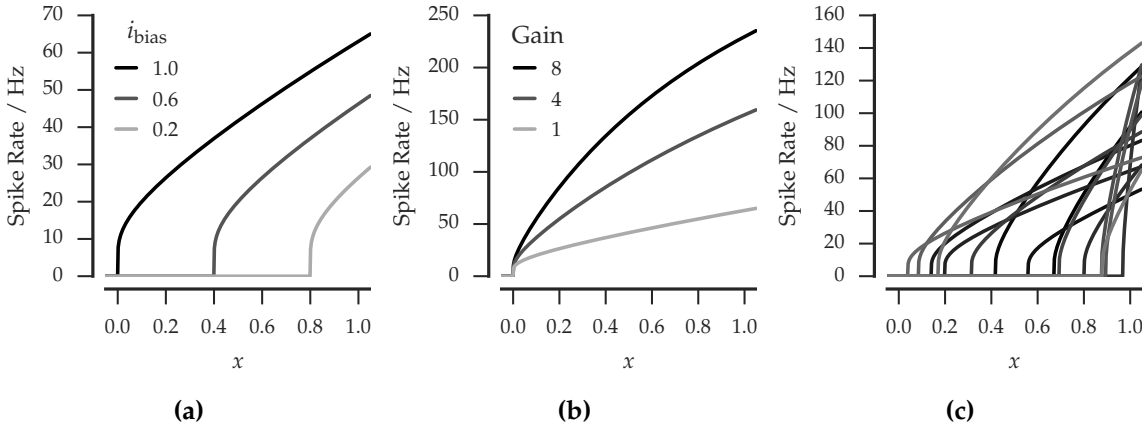


Figure 2.2 – Varying parameters of a LIF neuron with $\tau_{\text{ref}} = 2\,\text{ms}$ and $\tau_{\text{rc}} = 20\,\text{ms}$. In (a) the gain is held at 1 and the bias current, i_{bias}, is varied. In (b) the bias is held at 1 and the gain is varied. Finally, (c) illustrates how a range of tuning curves can be achieved through varying both the bias current and the gain.

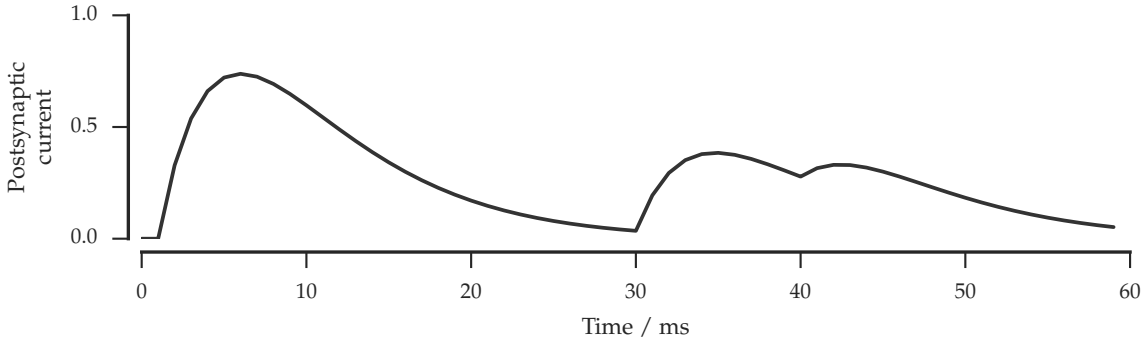


Figure 2.3 – Postsynaptic currents from three weighted and shaped spikes. Spikes occurred at times t =1 ms, 30 ms and 40 ms and had weights of 10, 5 and 2 units respectively.

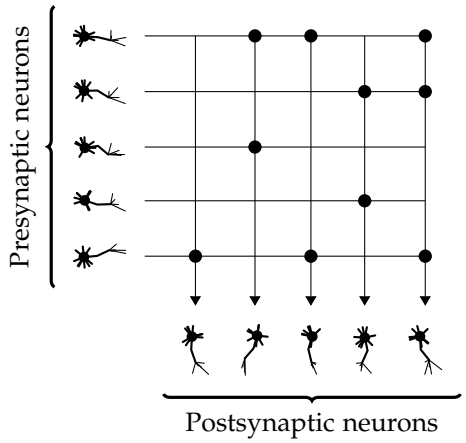


Figure 2.4 – Sample weight matrix between two populations of neurons

synaptic connections whose weights are selected to compute functions of the values represented by the transmitting neurons and whose synaptic shaping is leveraged to both smooth the spikes and implement system dynamics. By constraining the number and types of neurons and synapses in the model it is possible to compare the neural implementation of a particular functional description with the brain region thought to fulfil the same functional role. Consequently, hypotheses regarding the nature of some neural system (e.g., the role of basal ganglia in action-selection (Gurney, Prescott, and Redgrave 2001; Stewart, Choo, and Eliasmith 2010)) can be tested by implementing a model of the system and comparing aspects of the implementation (e.g., timing response, failure modes, etc.) with known biological values.

At the heart of the NEF are three principles:

Representation Populations of neurons act to represent *values* of the kind that we would use to typify the behaviour of a system (e.g., velocity).

Transformation Connections between ensembles can be used to *compute* functions of the values represented by the neurons (e.g., speed).

Dynamics *Synaptic filtering* of the spikes transmitted between ensembles may be used to implement dynamical systems (e.g., a recurrent connection can be used to construct an integrator which may be used to compute displacement).

Representation

The NEF characterises neural representation using two processes. *Encoding* converts the value of interest (e.g., a displacement, temperature, or other physical characteristic) into a spike-train. *Decoding* the spike-train returns an estimation of the original value. Alternatively expressed, the *encoding* process takes a low-dimensional value and transforms it into a high-dimensional space and the *decoding* process is able to recreate an estimation of the original value given the value in the higher-dimensional space.

Eliasmith and Anderson (2004) use a analogue-to-digital converter (ADC) to explain the processes of encoding and decoding. At its input the ADC receives, from a transducer, an analogue signal which represents some physical quantity. The ADC converts this input signal, which is continuous in both time and value, into a series of digital samples. These digital samples – constructed by a highly non-linear encoding – represent,

with some quantisation noise, the original value of interest but in the 'language' that is used to communicate between components of a digital rather than an analogue circuit. We can recover a representation of the original value by multiplying the ith bit of the output of the ADC by 2^i and summing, i.e., $\sum_{i=0}^{n-1} 2^i y[i]$ where y is the output of the ADC. This representation is a *decoding* of the value that is represented by the output and is an *approximation* of the input.

The same principles of encoding and decoding may be applied to a population of neurons. Each neuron acts as an encoder which converts its input into a series of spikes – the 'language' that is used to communicate between neurons. The timing characteristics of these spikes depends on the dynamics of the neuron and the selectivity of the neuron to the space it represents. This selectivity is achieved by adding an *encoder* term, e_i, to the expression for the input of a neuron:

$$i_{\text{input}} = g e_i x + i_{\text{bias}} \tag{2.2}$$

where g is the gain of the neuron and x is an input over which we have control (see p. 29).

When representing scalar values the encoder may be either $+1$ or -1 resulting in positive and negative sloping tuning curves respectively. For example, Figure 2.5 shows the tuning curves of two LIF neurons configured to act as an *on*/*off* pair. Their response to a time-varying input signal and the decoding of their spiking output are also shown. As can be seen, as the input signal (a 1 Hz sine wave) becomes increasingly positive the "on" neuron fires increasingly frequently; as the signal becomes increasingly negative the

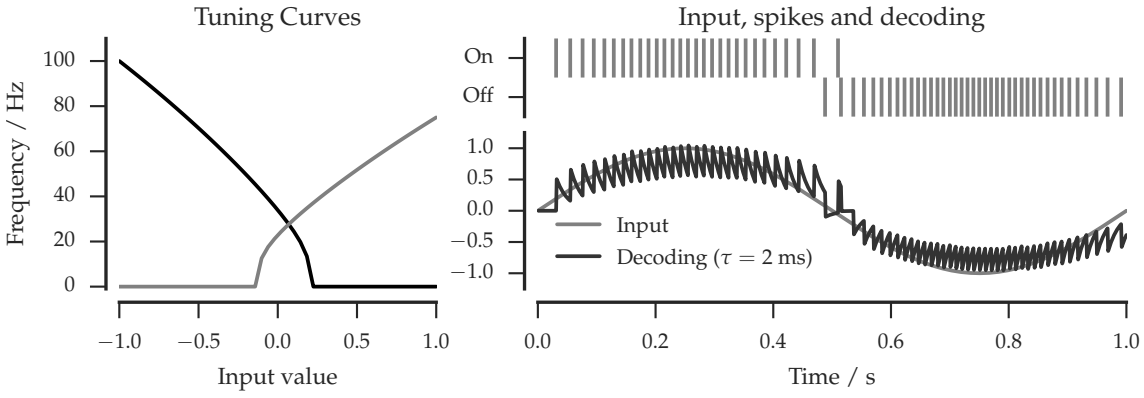


Figure 2.5 – Response of a pair of LIF neurons to a time-varying input signal. The tuning curves of the neurons, their spiking response to the input and the low-pass filtered linear decoding of the spiking output are shown.

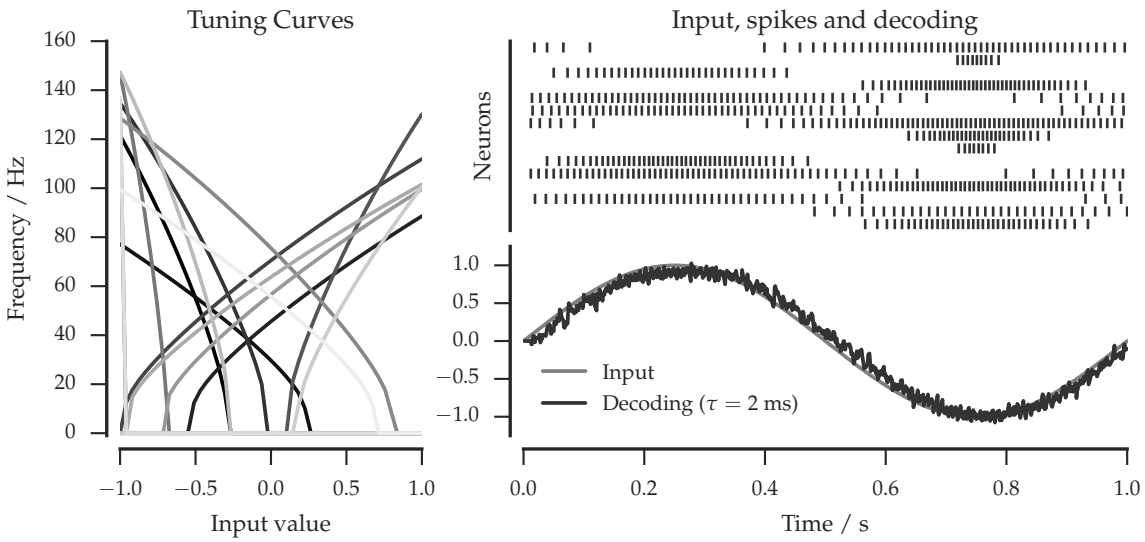


Figure 2.6 – Response of a population of 15 LIF neurons to a time-varying input signal. Note that the decoded representation of the ensemble is cleaner than that of Figure 2.5.

"off" neuron fires more strongly. By using more neurons we can increase the accuracy of the representation and decrease the noise present in the decoding. Figure 2.6 shows the response of a *population* or *ensemble* of 15 neurons to the same input signal, as each spike has a proportionally less effect the decoding is a 'cleaner' recreation of the input.

To *decode* the spike train emanating from a neural population we use a technique similar to that used to decode the output of the ADC. Each neuron is associated with a decoder d_i, and the decoding of the entire population is simply the sum of the spikes of the population multiplied by the decoders. This can be thought of as a matrix-vector multiply, as shown in Figure 2.7. The decoding is usually filtered to emulate the effects of synaptic transmission. Consequently, in the two neuron case (Figure 2.5) we see that as the "on" neuron fires more strongly the decoded output increases in value and as the "off" neuron fires more strongly the decoded value decreases.

Correct choice of decoder values is important; Stewart and Eliasmith (2014) describe a method to select such values automatically but a simple approach is outlined below. The tuning curves for the example 'on' and 'off' neurons can be approximated by the linear piece-wise functions, shown in left-hand pane of Figure 2.8:

$$a_{\text{On}}(x) = \begin{cases} 67x + 8 & -0.12 < x \\ 0 & x \leq -0.12 \end{cases} \qquad a_{\text{Off}}(x) = \begin{cases} 0 & 0.2 \leq x \\ -83x + 17 & x < 0.2 \end{cases}$$

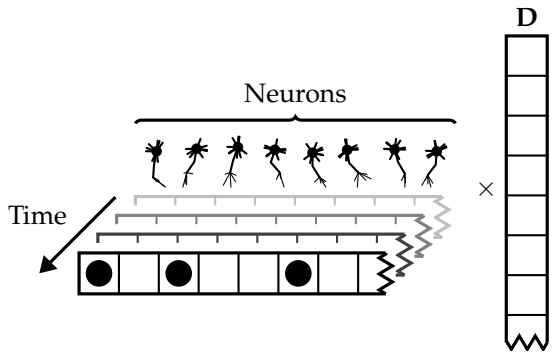


Figure 2.7 – Decoding the output of a population of neurons. For each discrete simulation time step a vector can be constructed whose elements indicate whether each neuron fired or not (these are shown as the stacked row vectors on the left of the figure). Multiplying this vector by a decoder matrix (**D**) results in a single value which is an estimate of the value represented by the population of neurons.

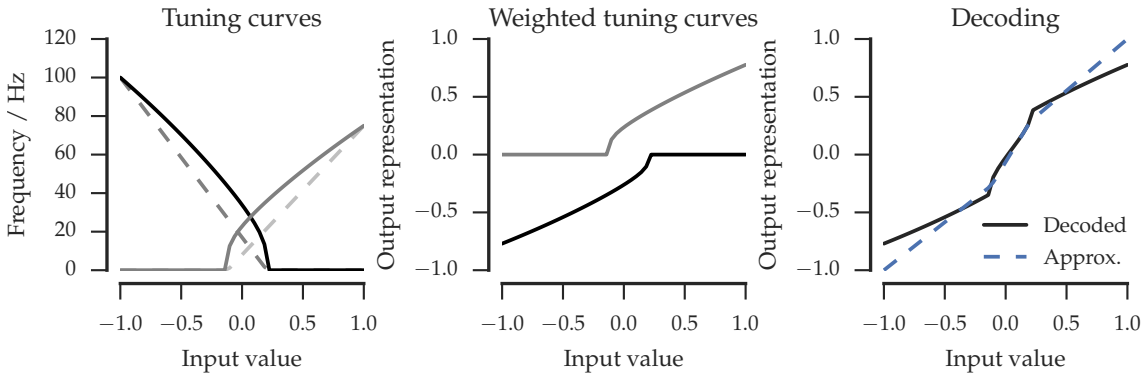


Figure 2.8 – Multiplying the tuning curves of a population by the decoders. The tuning curves and decoders are those from Figure 2.5. Piece-wise approximations of the tuning curves and the decoding formed from these approximations are shown as dashed lines.

We now want to find the decoders d_{On} and d_{Off} such that $\hat{x} = d_{\text{On}}a_{\text{On}}(x) + d_{\text{Off}}a_{\text{Off}}(x) \approx x$. One way to do this is to pick values of d_{On} and d_{Off} such that $\hat{x} = x$ for $x = 1$ and $x = -1$. For $x = 1$ we have $\hat{x} = d_{\text{On}}\,(67 + 8) = 1$, therefore $d_{\text{On}} = \frac{1}{75}$. At $x = -1$ we have $\hat{x} = d_{\text{Off}}\,(83 + 17) = -1$, therefore $d_{\text{Off}} = -\frac{1}{100}$.

Using these decoders gives:

$$d_{\text{On}}a_{\text{On}}(x) = \begin{cases} 0.9x + 0.1 & -0.12 < x \\ 0 & x \leq -0.12 \end{cases} \qquad d_{\text{Off}}a_{\text{Off}}(x) = \begin{cases} 0 & 0.2 \leq x \\ 0.8x - 0.2 & x < 0.2 \end{cases}$$

Resulting in the decoding:

$$d_{\text{On}}a_{\text{On}} + d_{\text{Off}}a_{\text{Off}} = \begin{cases} 0.9x + 0.1 & 0.2 \leq x \\ 1.7x - 0.1 & -0.12 < x < 0.2 \\ 0.8x - 0.2 & x \leq -0.12 \end{cases}$$

Note that this decoding curve does not exactly match the desired function $f(x) = x$ but approximates it closely aside from a 'bump' in the range $-0.12 < x < 0.2$. This decoding and a similar 'bump' in the decoding formed from the tuning curves, rather than from the linear approximations, are shown in Figure 2.8. As will be seen in Section 2.2 decoders can be chosen such that the decoding is a transformation of the input value.

If the *encoders*, $\boldsymbol{e}_i$, and *decoders* are selected to be vectors rather than scalars a population may be used to represent a multi-dimensional value, the vector $\boldsymbol{x}$. The input current to a neuron is now derived as $i_{\text{input}} = g\boldsymbol{e}_i \cdot \boldsymbol{x} + i_{\text{bias}}$, such that neurons fire more strongly the more similar the represented value is to their encoding vector. An example of this is shown in Figure 2.9 where four neurons are used to represent a 2-dimensional value. Each neuron in the ensemble is receptive to a portion of the representational space.

Figure 2.9(a) shows how the firing rate of a neuron with the encoder $\boldsymbol{e} = \left(-\frac{1}{\sqrt{2}}, +\frac{1}{\sqrt{2}}\right)$ varies with the input representation. Described on the *tuning surface* of this neuron is a circle, with a radius of one, centred at the origin. In Figure 2.9(b) the firing rates on the perimeter of this circle are redrawn against their angular position, along with the tuning curves of a further three neurons. Finally, Figure 2.9(c) shows how, together, these four neurons can be used to represent a two dimensional value. As before, greater accuracy could be achieved through using more neurons both because each spike will have

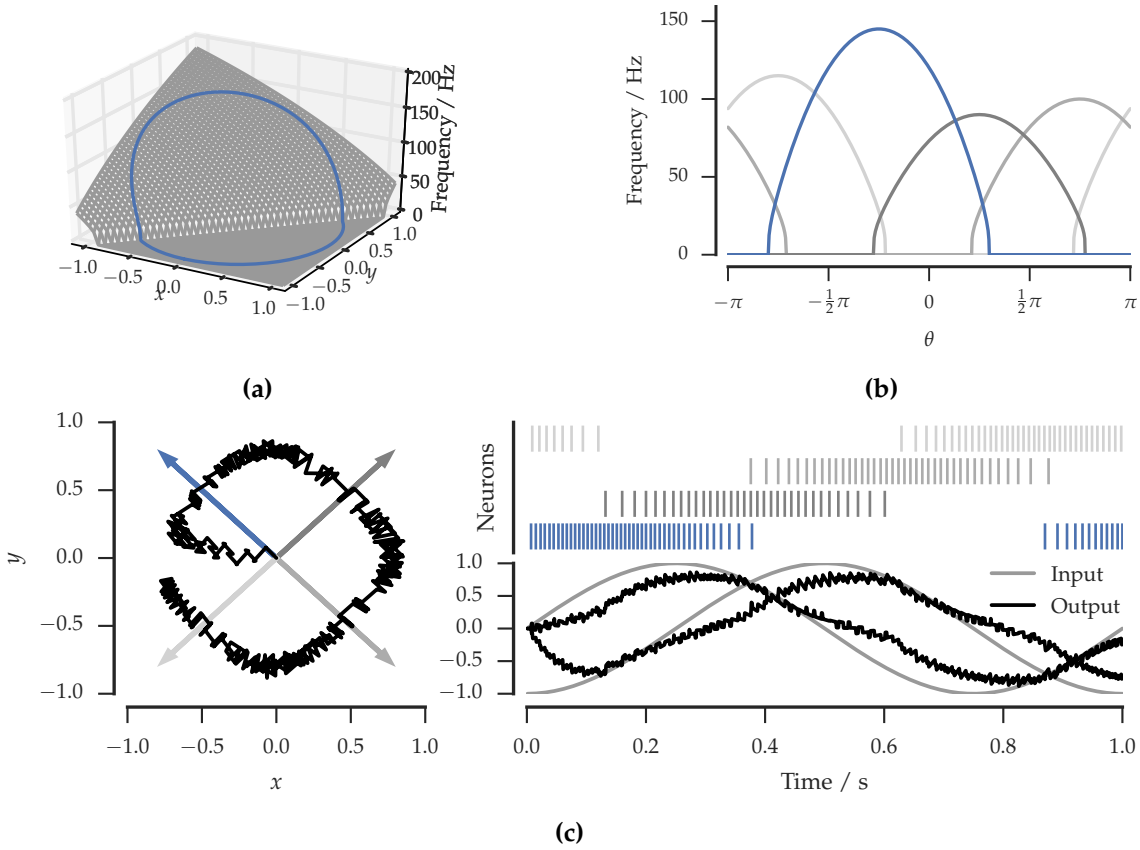


Figure 2.9 – Representing a 2D value with four neurons. The tuning *surface* of a single neuron in shown in (a) – described on this surface is a circle of radius one centred at the origin. In (b) the firing rates on the perimeter of this circle are plotted against their angular position and the responses of a further three neurons are shown. This population can be used to represent a 2D value, as shown in (c) – note that since at time $t = 0$ the neurons are not firing there is a transient in the output.

proportionally less effect but also because the encoders will better 'cover' the space. All that is required to represent higher dimensional spaces, such as those used in Spaun, are higher dimensional encoders.

Thus far we have seen how to use a population of neurons to convert a scalar or vector value into a number of spike trains and how a linear decoding of these spikes can be used to form an approximation of the original scalar or vector. However, biological neural networks contain many interconnected populations of neurons, between which connections are described by synaptic weight matrices. Using the NEF we can construct a weight matrix, ω, as the product between the encoders of the post-synaptic population,

E, and the decoders of the pre-synaptic population, **D**.

$$\omega = \mathbf{ED} \tag{2.3}$$

Imagine we were to connect our two neuron population from before (Figure 2.5 and Figure 2.8) with the decoders $\mathbf{D} = \begin{pmatrix} d_{\text{On}} & d_{\text{Off}} \end{pmatrix} = \begin{pmatrix} \frac{1}{75} & -\frac{1}{100} \end{pmatrix}$ to another pair of neurons. We let this other population consist of another pair of 'on' and 'off' neurons with the encoders:

$$\mathbf{E} = \begin{pmatrix} +1 \\ -1 \end{pmatrix}$$

To compute the synaptic weight matrix of the connection between these populations we multiply the encoders of the post-synaptic matrix, **E**, by the decoders of the pre-synaptic matrix, **D**.

$$\omega = \begin{pmatrix} +1 \\ -1 \end{pmatrix} \begin{pmatrix} \frac{1}{75} & -\frac{1}{100} \end{pmatrix} = \begin{pmatrix} \frac{1}{75} & -\frac{1}{100} \\ -\frac{1}{75} & \frac{1}{100} \end{pmatrix}$$

The matrix ω is constructed such that the first column describes connections *from* the presynaptic-'on' neuron and the second describes connections *from* the presynaptic-'off' neuron. Additionally, the first and second rows describe connections *to* the postsynaptic-'on' and -off neurons respectively. Consequently we note that there are *excitatory* connections between the 'on' neurons and between the 'off' neurons: $\omega_{\text{On,On}}$ and $\omega_{\text{Off,Off}}$ are positive. In addition to which there are *inhibitory* connections from the presynaptic-'on' to the postsynaptic-'off' neuron and from the presynaptic-'off' neuron to the postsynaptic-'on' neuron: $\omega_{\text{On,Off}}$ and $\omega_{\text{Off,On}}$ are negative. As a result, when the presynaptic population represents a positive value (causing the presynaptic-'on' neuron to become active) the postsynaptic-'on' will be excited and the postsynaptic-'off' neuron will be inhibited. Likewise, when the presynaptic population represents a negative value (causing the presynaptic-'off' neuron to become active) the postsynaptic-'off' neuron will be excited and the postsynaptic-'on' neuron inhibited.

An example with vector (rather than scalar) encoders and decoders gives a better intuition of the meaning of the synaptic weight matrix. Consider an example consisting of two ensembles, consisting of three and four neurons respectively, representing a two

dimensional space. The encoders of the postsynaptic population, $\boldsymbol{e}_i$, and the decoders of the presynaptic population, $\boldsymbol{d}_j$, are:

$$\begin{aligned}
\boldsymbol{e}_1 &= \begin{pmatrix} -0.6 & 0.8 \end{pmatrix} & \boldsymbol{d}_1 &= \begin{pmatrix} -0.2 & 0.45 \end{pmatrix}^\top \\
\boldsymbol{e}_2 &= \begin{pmatrix} -0.9 & -0.4 \end{pmatrix} & \boldsymbol{d}_2 &= \begin{pmatrix} -2.3 & -0.6 \end{pmatrix}^\top \\
\boldsymbol{e}_3 &= \begin{pmatrix} -0.9 & -0.2 \end{pmatrix} & \boldsymbol{d}_3 &= \begin{pmatrix} 2.1 & 0.9 \end{pmatrix}^\top \\
\boldsymbol{e}_4 &= \begin{pmatrix} 0.3 & 0.9 \end{pmatrix}
\end{aligned}$$

We intend to select the synaptic weight matrix such that the firing rate of a neuron in the second population is proportional to the value being represented by the output of the first population. A simple way to achieve this is to set the synaptic weight between every pair of pre- and postsynaptic neurons according to the similarity between their respective decoders and encoders. For example, the synaptic weight between the second neuron of the first population and the third neuron of the second will be given by:

$$\omega_{3,2} = \boldsymbol{e}_3 \boldsymbol{d}_2 = (-0.9 \cdot -2.3) + (-0.2 \cdot -0.6) = 2.19$$

Forming the encoders and decoders into matrices (**E** and **D**, respectively) and multiplying them together forms the complete weight matrix:

$$\omega = \mathbf{ED} = \begin{pmatrix} \boldsymbol{e}_1 \\ \boldsymbol{e}_2 \\ \boldsymbol{e}_3 \\ \boldsymbol{e}_4 \end{pmatrix} \begin{pmatrix} \boldsymbol{d}_1 & \boldsymbol{d}_2 & \boldsymbol{d}_3 \end{pmatrix} = \begin{pmatrix} \boldsymbol{e}_1\boldsymbol{d}_1 & \boldsymbol{e}_1\boldsymbol{d}_2 & \boldsymbol{e}_1\boldsymbol{d}_3 \\ \boldsymbol{e}_2\boldsymbol{d}_1 & \boldsymbol{e}_2\boldsymbol{d}_2 & \boldsymbol{e}_2\boldsymbol{d}_3 \\ \boldsymbol{e}_3\boldsymbol{d}_1 & \boldsymbol{e}_3\boldsymbol{d}_2 & \boldsymbol{e}_3\boldsymbol{d}_3 \\ \boldsymbol{e}_4\boldsymbol{d}_1 & \boldsymbol{e}_4\boldsymbol{d}_2 & \boldsymbol{e}_4\boldsymbol{d}_3 \end{pmatrix} = \begin{pmatrix} 0.48 & 0.90 & -0.54 \\ 0 & 2.31 & -2.25 \\ 0.09 & 2.19 & -2.07 \\ 0.35 & -1.23 & 1.44 \end{pmatrix}$$

This procedure works for any pair of neural populations whose encoders and decoders can be formed into a matrices of shape $N_{\text{post}} \times D$ and $D \times N_{\text{pre}}$ and will always result in a synaptic weight matrix of form $N_{\text{post}} \times N_{\text{pre}}$.

It should be noted that, since a pair of neurons will only *not* be connected if their encoders and decoders are orthogonal (for example, $\omega_{2,1}$ above), the synaptic weight matrices produced this way will be *dense*. This has important ramifications for the sim-

ulation of these networks. In addition, the weight matrices often violate the observation that, in biology, neurons only ever form either *excitatory* or *inhibitory* outgoing connections (Dale's Principle, see Strata and Harvey 1999). Eliasmith and Anderson (2004) and Tripp and Eliasmith (2016) describe methods to build networks such that they obey this principle.

Transformation

In the previous section we saw that the decoders for a population of neurons could be chosen to allow us to estimate the value represented by the ensemble. However, it is also possible to choose decoders to estimate *a function of* the value represented by a population of neurons. For example, we might select decoders such that we could estimate the *square* of the value represented by an ensemble.

We can approximate this process for the two neurons configured as an *on/off* pair. Using the same tuning curve approximations as before we know that we would like to find d^f_{On} and d^f_{Off} such that $d^f_{\text{On}}a_{\text{On}} + d^f_{\text{Off}}a_{\text{Off}} \approx f(x)$, where $f(x) = x^2$. Keeping $d^f_{\text{On}} = \frac{1}{75}$ as before, but negating d^f_{Off} to be $+\frac{1}{100}$ results in the decoding:

$$d^f_{\text{On}}a_{\text{On}} + d^f_{\text{Off}}a_{\text{Off}} = \begin{cases} 0.9x + 0.1 & 0.2 \leq x \\ 0.1x + 0.3 & -0.12 < x < 0.2 \\ -0.8x + 0.2 & x \leq -0.12 \end{cases}$$

which approximates the form of x^2. Figure 2.10 shows how the use of these decoders can compute a different function from the same spiking output that was shown in Figure 2.5. Once again, with more neurons a 'cleaner' decoding may be achieved (as in Figure 2.11).

These new decoders $\mathbf{D}^f = \begin{pmatrix} \frac{1}{75} & \frac{1}{100} \end{pmatrix}$, may be used to form the synaptic weight matrix specifying the connection to another pair of 'on' and 'off' neurons. Using the same encoders as previously, $\mathbf{E} = \begin{pmatrix} +1 & -1 \end{pmatrix}^\top$, results in the synaptic weight matrix:

$$\omega^f = \begin{pmatrix} +1 \\ -1 \end{pmatrix} \begin{pmatrix} \frac{1}{75} & \frac{1}{100} \end{pmatrix} = \begin{pmatrix} \frac{1}{75} & \frac{1}{100} \\ -\frac{1}{75} & -\frac{1}{100} \end{pmatrix}$$

The weight matrix is such that *regardless* of the value represented by the presynaptic population – i.e., whichever of the presynaptic-'on' or the presynaptic-'off' neuron is

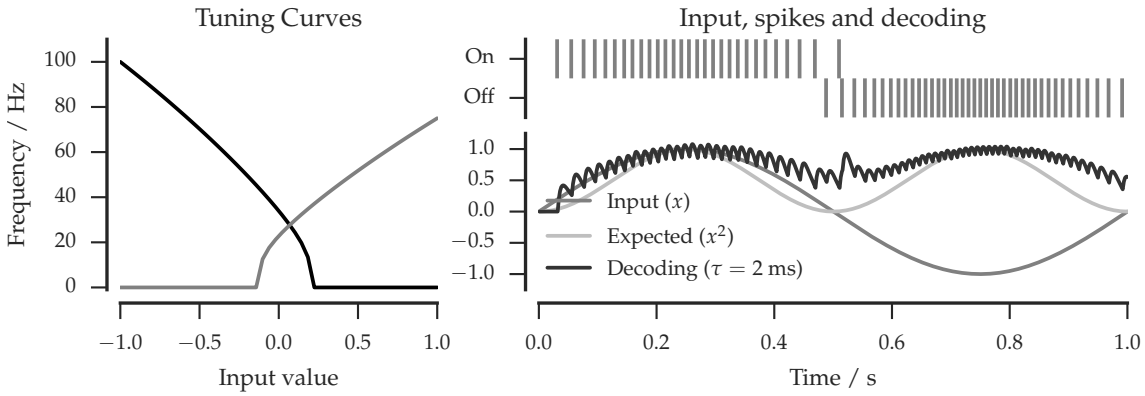


Figure 2.10 – Decoding x^2 from a pair of neurons. The left panel shows the tuning curves of the two neurons. The input to the neurons, the resulting spikes and a decoding of the spikes which reproduces an estimate of the square of the input are shown in the right panel.

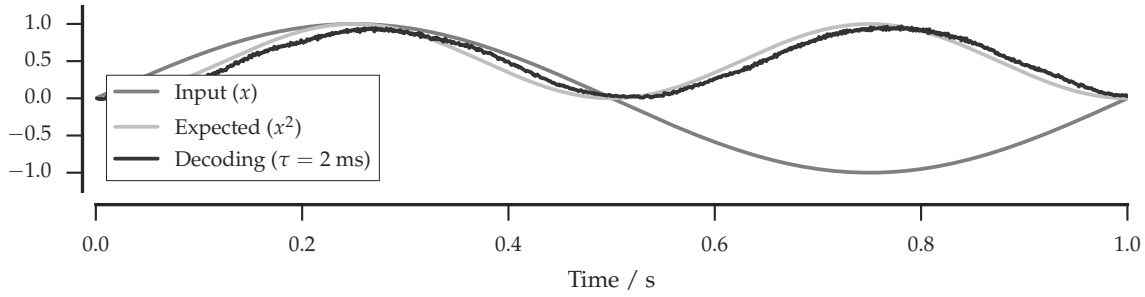


Figure 2.11 – Decoding the square of the value represented by an ensemble of 100 neurons. The input to the ensemble, the expected output (the square of the input) and the measured output are shown. Note that the output from the ensemble is both less noisy and a closer approximation of the value x^2 than that produced by the smaller ensemble in Figure 2.10.

firing – the postynaptic-'on' neuron will be excited and the presynaptic-'off' neuron will be inhibited. Since the decoders of the presynaptic population were selected to compute the square of the value represented by the ensemble, the result of which is never negative, it is not surprising that the weights in the synaptic matrix act to excite the postsynaptic-'on' neuron and inhibit the postsynaptic-'off' neuron.

Instead of computing a function of the value represented by an ensemble we may wish to transform it with a linear operator, $\mathbf{L}$. For example, we might wish to rotate the 2-dimensional value represented by an ensemble by the angle θ. This may be achieved by premultiplying the decoders by the desired transform:

$$\mathbf{L}\boldsymbol{d}_i = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \boldsymbol{d}_i \tag{2.4}$$

Figure 2.12 shows how a linear operator can 'rotate' the value decoded from an ensemble.

We may combine this technique with the previous example. Suppose that instead of decoding x^2 from an ensemble we wished to decode $-\frac{1}{2}x^2$ we could modify our previous decoders $\mathbf{D}^f = \begin{pmatrix} \frac{1}{75} & \frac{1}{100} \end{pmatrix}$ by premultiplying them by $-\frac{1}{2}$ to result in $\mathbf{D}^g = \begin{pmatrix} -\frac{1}{150} & -\frac{1}{200} \end{pmatrix}$. As before, multiplying by the encoders allows us to create a synaptic weight matrix between two pairs of neurons which results in the value $-\frac{1}{2}x^2$ being applied to the postsy-

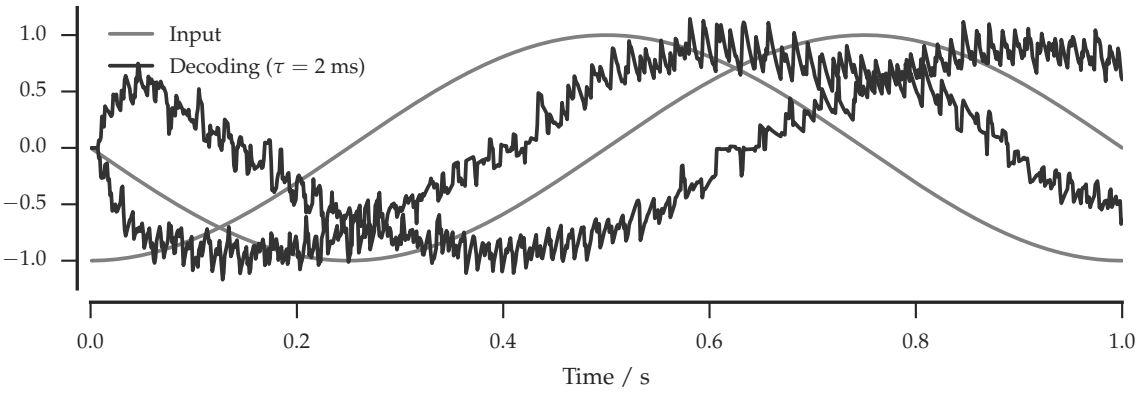


Figure 2.12 – Applying a linear transform to the decoding of an ensemble. Premultiplying the decoders of an ensemble by a linear transform applies that transform to the decoding of the ensemble – in this case a $\frac{1}{4}\pi$ rotation has been applied.

naptic population:

$$\omega^g = \begin{pmatrix} +1 \\ -1 \end{pmatrix} \begin{pmatrix} -\frac{1}{150} & -\frac{1}{200} \end{pmatrix} = \begin{pmatrix} -\frac{1}{150} & -\frac{1}{200} \\ \frac{1}{150} & \frac{1}{200} \end{pmatrix}$$

Unlike the weight matrix, w^f, derived using decoders for x^2, this weight matrix causes the postsynaptic-'off' neuron to be excited and the postsynaptic-'on' neuron to be inhibited regardless of the value represented by the presynaptic pair. Since $-\frac{1}{2}x^2$ is never positive this pattern of connectivity is to be expected.

Dynamics

Synaptic connections between neurons are characterised not only by the weighting but also by the *shaping* they impart to the spikes which they carry. This shaping can be modelled as a linear transfer function. By exploiting the presence of this transfer function a number of dynamic systems may be implemented neurally (Eliasmith 2005; Eliasmith and Anderson 2004).

Figure 2.13(a) shows a simple neural network consisting of a single, recurrently connected, ensemble fed by the value $u(t)$ and representing the value $x(t)$. Assuming that the synaptic filter model $h(t)$ is used for both the connection into the ensemble and the recurrent connection, and that the linear transform $\mathbf{L}$ is applied by the recurrent connection then this neural system may be approximated by the block diagram shown in Figure 2.13(b). Evaluating the block diagram results in $x(t) = h(t) * u(t) + h(t) * \mathbf{L}x(t)$. Or, in the Laplace domain:

$$\begin{aligned} X(s) &= H(s)U(s) + H(s)\mathbf{L}X(s) \\ &= H(s)\left(U(s) + \mathbf{L}X(s)\right) \end{aligned}$$

Letting $H(s) = \frac{1}{\tau s+1}$, the transfer function for a first-order low-pass filter with time constant τ, results in:

$$\begin{aligned} X(s) &= \frac{U(s) + \mathbf{L}X(s)}{\tau s + 1} \\ \tau s X(s) + X(s) &= U(s) + \mathbf{L}X(s) \\ sX(s) &= \frac{U(s) + \mathbf{L}X(s) - X(s)}{\tau} \end{aligned}$$

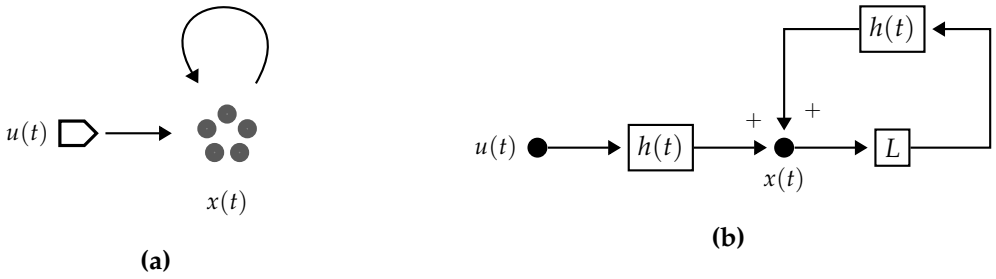


Figure 2.13 – Using a recurrent connection to implement a dynamic system. A recurrently connected ensemble representing $x(t)$ is shown in (a), alongside the recurrent connection the ensemble is being fed the value $u(t)$. If the synaptic filter model applied on both connections in (a) is $h(t)$ then an approximate block diagram for the system is as shown in (b). The use of a small number of clustered dots to represent a neural ensemble is common to the field and will be reused throughout this thesis.

Which, after applying the inverse Laplace transform, is:

$$\dot{x} = \frac{u(t) + (\mathbf{L} - \mathbf{I})\, x(t)}{\tau} \tag{2.5}$$

Consequently, through choice of $\mathbf{L}$ we can determine the dynamics of x. For example, to create an integrator with dynamics $\dot{x}(t) = v(t)$ (where v and x are scalars) we could let $\mathbf{L} = 1$ and $u(t) = \tau v(t)$:

$$\dot{x} = \frac{u(t) + (\mathbf{L} - 1)\, x(t)}{\tau} = \frac{\tau v(t) + (1-1)x(t)}{\tau} = v(t)$$

A simulation of this network, with $\tau = 100\,\text{ms}$, is shown in Figure 2.14. Attractor networks like this can be used to build memories to store vectors, which, as shown in the next section, can be used to represent *token*-like values.

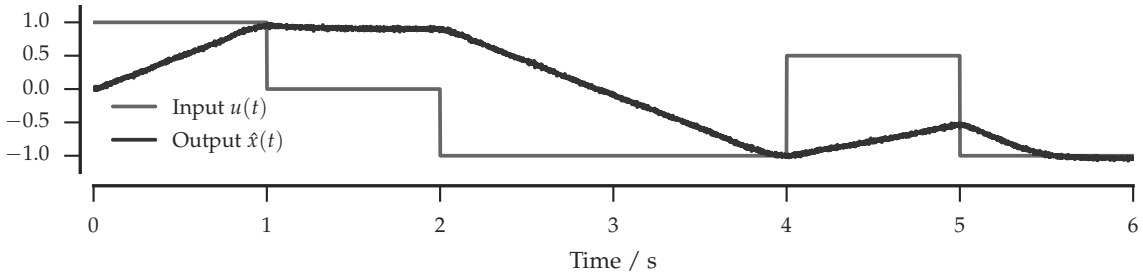


Figure 2.14 – Simulation of a neurally implemented integrator. The neural network simulated here is as presented in Figure 2.13.

2.3 The Semantic Pointer Architecture

Using the Neural Engineering Framework it is possible to construct neural implementations of control-theory like structures. However, not all cognitive processes are best described by dynamic systems. For example, language consists of *tokens* (words) which are combined together to form more structured tokens (phrases and sentences). Since *symbolic*, computational-like, processing seems to be so important in human cognition it has long been argued it must be accounted for in any theory of cognition. Unfortunately, mapping structured symbolic representations to neural networks is non-trivial (Smolensky 1990) and has been hotly debated (Fodor and Pylyshyn 1988; Smolensky 1988).

In the Semantic Pointer Architecture (SPA) (Eliasmith 2013; Eliasmith, Stewart, et al. 2012) each *symbol*, such as the word 'cheese', is represented by a vector in a high-dimensional space. By combining different vectors together with different operators new, structured, symbols may be formed and computations performed. Since the Neural Engineering Framework describes how vectors could be represented and transformed by neural networks the combination of the SPA and NEF presents a method by which symbol-like processing can be implemented in neural form.

Representing and operating on symbols

Vectors are a natural way to represent symbols since they support inference of *semantic relationships*: symbols which are semantically similar (e.g., *horse* and *donkey*) may be assigned vectors which are close within the high dimensional space. By using a sufficiently high dimensional space, spurious relationships between vectors can be avoided since the likelihood of any two vectors being similar is vanishingly small. Relatedness between a pair of vectors-representing-symbols can be measured by computing their dot product.

Vector addition is one method by which symbols in a vector symbolic architecture may be combined. Since addition preserves *relatedness* it is ideal for producing new symbols. For example, the vector representing *pet fish*, $\mathbf{PET} + \mathbf{FISH}$, could be expected to have similarities with the vector representing *pet mice*, $\mathbf{PET} + \mathbf{MICE}$. However, addition is not good at constructing *syntactic representations* of the form *mice eat cheese*. This symbol could be expressed as $\mathbf{MICE} + \mathbf{EAT} + \mathbf{CHEESE}$ but this vector could not be differentiated from that which would be used to represent the symbol *cheese eat mice*. What is needed, in addition to the addition operator, is a way to *bind* symbols together to pro-

duce new, distinct, symbols. In the Semantic Pointer Architecture this is achieved by use of *circular convolution* (Plate 1995).

Circular convolution of two vectors, $\boldsymbol{a} \circledast \boldsymbol{b}$, results in a vector, of the same dimensionality, which is highly dissimilar from either of the original vectors, $\boldsymbol{a}$ or $\boldsymbol{b}$, but which contains sufficient information that either of the originals may be retrieved given some information about the other. Specifically, the circular convolution of two n-dimensional vectors, $\boldsymbol{a}$ and $\boldsymbol{b}$ is defined as:

$$(\boldsymbol{a} \circledast \boldsymbol{b})_i = \sum_{j=0}^{n-1} \boldsymbol{a}_j \boldsymbol{b}_{i-j} \mod n \tag{2.6}$$

which, due to the convolution theorem, may also be expressed as:

$$\boldsymbol{a} \circledast \boldsymbol{b} = \mathcal{F}^{-1}\left(\mathcal{F}\boldsymbol{a} \odot \mathcal{F}\boldsymbol{b}\right) \tag{2.7}$$

where, $\mathcal{F}$ is the discrete Fourier transform, $\mathcal{F}^{-1}$ its inverse and $\odot$ the element-wise multiplication operator.

Using circular convolution to bind *placeholder* tokens (e.g., **SUBJ** for *subject*, **OBJ** for *object* and **VRB** for *verb*) to the constituents of a sentence is one way to construct syntactic representations. Doing this allows us to express *mice eat cheese* as

$$\mathbf{S} = \mathbf{MICE} \circledast \mathbf{SUBJ} + \mathbf{EAT} \circledast \mathbf{VRB} + \mathbf{CHEESE} \circledast \mathbf{OBJ}$$

which is clearly distinct from the expression which would represent *cheese eat mice*.

For the following example the symbols **MICE**, **EAT**, **CHEESE**, **SUBJ**, **OBJ** and **VRB** were all represented by randomly selected, 128 dimension, vectors of length 1. The compound expression **S** contains pairs of these symbols bound together using the circular convolution operation defined above. Each of these bound pairs is strongly dissimilar from the original symbols from which it was formed. Table 2.1 shows the dot product similarity between the compound symbols composing **S** and each of the original symbols from the vocabulary. Since **S** was formed as the summation of the bound pairs it is similar to each of them.

Symbols can be *unbound* from a compound symbol to retrieve information from a complex representation. To unbind a symbol we construct a pseudo-inverse of the sym-

	MICE	**EAT**	**CHEESE**	**SUBJ**	**OBJ**	**VRB**	**S**
MICE ⊛ **SUBJ**	−0.03	0.05	0.03	0.00	0.10	0.01	0.60
EAT ⊛ **VRB**	0.07	−0.19	0.08	−0.14	−0.03	0.09	0.61
CHEESE ⊛ **OBJ**	0.10	−0.02	−0.18	−0.08	0.12	−0.03	0.58
S	0.08	−0.09	−0.04	−0.12	0.11	0.04	1.00

Table 2.1 – Similarity between compound symbols and their constituents. Each compound symbol, **A** ⊛ **B**, is also represented by a 128-dimensional vector which has little similarity (as measured by dot product) to either of its constituent symbols.

bol and bind it with the compound symbol using the circular convolution operator. For example, to unbind **SUBJ** from **MICE** ⊛ **SUBJ** we would construct the inverse symbol $\mathbf{SUBJ}^*$ and bind it with the compound symbol, **MICE** ⊛ **SUBJ** ⊛ $\mathbf{SUBJ}^*$. The pseudo-inverse (with respect to circular convolution) of an n-dimensional vector $\boldsymbol{a}$ can be constructed as $\boldsymbol{a}^* = (\boldsymbol{a}_0, \boldsymbol{a}_{n-1}, \boldsymbol{a}_{n-2}, \ldots, \boldsymbol{a}_1)$. However, since this is only an *pseudo*-inverse the result of $(\mathbf{MICE} \circledast \mathbf{SUBJ}) \circledast \mathbf{SUBJ}^*$ is not exactly **MICE**. Plate (1995, §II.E) provides a demonstration of why the pseudo-inverse has this shape.

This same technique can be used to unbind terms from compound expressions that were constructed through the addition of bound symbols. For example, to determine the subject of the expression **S**, *mice eat cheese*, we would compute:

$$\mathbf{S} \circledast \mathbf{SUBJ}^* \approx \mathbf{MICE}$$

$$(\mathbf{MICE} \circledast \mathbf{SUBJ} + \mathbf{EAT} \circledast \mathbf{VRB} + \mathbf{CHEESE} \circledast \mathbf{OBJ}) \circledast \mathbf{SUBJ}^* \approx \mathbf{MICE}$$

Table 2.2 shows the similarity between the result of unbinding different symbols from **S** and the original symbols from the vocabulary. It is clear that each of the unbound symbols, $\mathbf{S} \circledast \mathbf{x}^*$, is most similar to the symbol that was originally bound with **x** – for example, $\mathbf{S} \circledast \mathbf{SUBJ}^*$ is most similar to **MICE**.

Implementations of this form of vector-based representation often replace the output of an unbinding operation with a canonical symbol from the vocabulary to reduce the

	MICE	**EAT**	**CHEESE**	**SUBJ**	**OBJ**	**VRB**	**S**
S ⊛ $\mathbf{SUBJ}^*$	0.51	0.02	−0.03	−0.16	0.05	0.03	−0.26
S ⊛ $\mathbf{VRB}^*$	−0.10	0.52	−0.14	0.04	−0.05	0.12	0.06
S ⊛ $\mathbf{OBJ}^*$	0.01	−0.07	0.50	0.05	−0.18	−0.04	−0.20
S ⊛ $\mathbf{MICE}^*$	−0.27	0.04	0.03	0.49	0.01	−0.10	0.07

Table 2.2 – Similarity of unbound symbols to vectors in the original vocabulary.

noise presented to later stages of computation. For example, since the result of unbinding **SUBJ** from **S** is most similar to **MICE** (the dot product was 0.51, see the first row of Table 2.2) the vector **MICE** will be presented to further stages of computation rather than the value of $\mathbf{S} \circledast \mathbf{SUBJ}^*$. In the Semantic Pointer Architecture this form of 'cleanup' is achieved with an autoassociative memory (Stewart, Tang, and Eliasmith 2011).

The 'question answering' example used above that extracted the subject, object and verb from the expression **S** can be implemented as a neural network using the principles of the Neural Engineering Framework. Figure 2.15 shows the top-level view of such a network. One input to the network is used to store the expression **S** in a memory and the other provides a series of *cues*. Each cue is inverted and fed to a neural network which implements circular convolution. Output from the memory is also fed to the circular convolution network to produce the 'answer' to the 'question' provided by the *cue*.

Figure 2.16 shows the results of a simulation of this network. The top panel of the figure shows the dot product between the input cue and each of the symbols **SUBJ**, **VRB** and **OBJ**. No cues were provided for the first 0.2 s of the simulation, during which the expression **S** was stored in the memory. Subsequently, each of the cues was presented for a period of 0.2 s. The lower panel shows the dot products between the output of the network and each of the symbols **MICE**, **EAT** and **CHEESE**. As expected, the output of the network was most similar to **MICE** when given the cue **SUBJ**, to **EAT** when given the cue **VRB** and to **CHEESE** when given the cue **OBJ**.

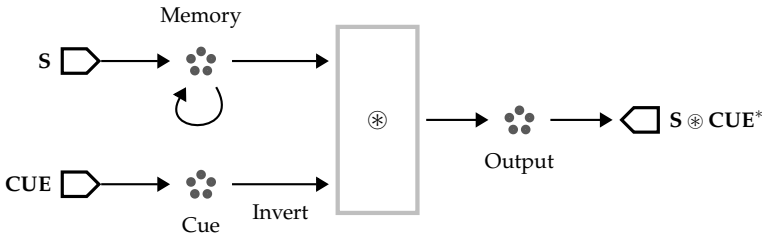


Figure 2.15 – High-level view of a symbol unbinding neural network. The structure of the circular convolution network (marked '⊛') is described later in the thesis.

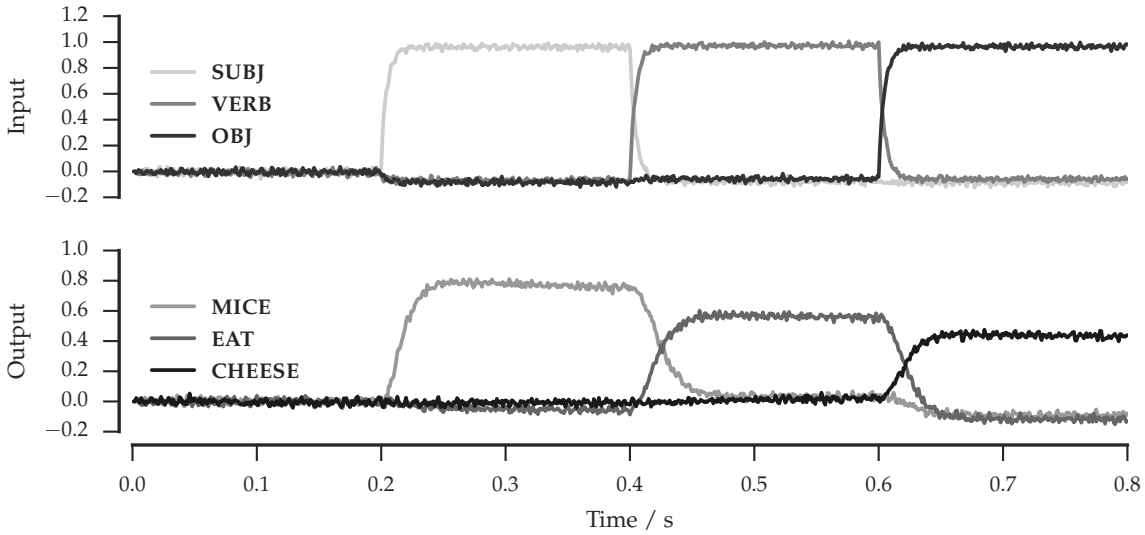


Figure 2.16 – Simulation of a neural implementation of symbol unbinding. The upper panel shows the input cues being applied to the neural network and the lower panel shows the effect of unbinding each cue against an expression, **S**, stored in a memory. Each line represents the time-varying similarity between a symbol in the vocabulary and the input or output value of interest.

Symbols and compressed representations

In the previous example we used a number of randomly selected vectors to represent base symbols (**SUBJ** for *subject* and so on). These base vectors were combined using addition and circular convolution to construct new vectors *of the same size* to represent new symbols. Necessarily these new vectors are somewhat *compressed* relative to their original constituents, evidence of which can be seen in the levels of dissimilarity found when unbinding base symbols from a compound symbol. However, new vectors may be formed not only through the combination and compression of existing vectors of the same dimensionality but also through the compression of vectors of higher dimensionality. Tang and Eliasmith (2010) extend the work of Hinton and Salakhutdinov (2006) to show that Deep Belief Networks (DBNs) can be used to construct compressed representations of inputs such as sensory stimuli. Consequently, some symbols in the Semantic Pointer Architecture may be represented by vectors which are derived from the characteristics of the referent of the symbol in some external modality such as vision or sound. Since SPA uses the dot product of vectors to determine their similarity, using external stimuli to form the vectors that are the input of a neural system is a reasonable way to ensure that similar input stimuli are translated into functionally similar vectors.

2.4 Cognitive architectures and Spaun

Section 2.1 described, and provided simple models of, the basic components of the nervous system. In Section 2.2, these components were combined into larger units which could represent and transform multi-dimensional values. Finally, methods for using multi-dimensional values to represent symbol-like values and for using these symbols in computation-like processes were introduced in Section 2.3. Each of these sections represented a different level of scale and complexity – from basic components through to complex representations. A similar story would take us from transistors to logic gates to combinatorial and sequential logic, and finally to processors.

Just as a processor includes different components which, combined, allow it to execute programs, cognitive architectures are constructed from the kinds of components that are expected to be required by a *cognitive* system: working and long term memory, attentional control, symbol manipulation etc. Each of these components or *modules* is proposed to represent the *function* of a certain brain region or regions and the role of the entire architecture is to explain how these components interact to produce *behaviour*. The main components of the Spaun cognitive architecture (Eliasmith, Stewart, et al. 2012) and their interconnection are pictured in Figure 2.17.

Action selection and execution

The combination of modules to produce behaviour can be illustrated by constructing a simple neural machine (illustrated in Figure 2.18) whose behaviour can be interpreted as repeated transitions among the states **A**, **B** and **C**. In practice, the current state of the system will be represented by a population of neurons representing the vector $\boldsymbol{x}$, which can take any value. We will consider the system to be *in* the *state* **A** when $\boldsymbol{x}$ is most like a vector we have selected to represent **A**, that is, when $\boldsymbol{a} \cdot \boldsymbol{x} > \boldsymbol{b} \cdot \boldsymbol{x}$ and $\boldsymbol{a} \cdot \boldsymbol{x} > \boldsymbol{c} \cdot \boldsymbol{x}$ (likewise for the other states). The system will be configured such that when it is 'in' state **A** the input of the population of neurons representing $\boldsymbol{x}$ will be changed to make $\boldsymbol{x}$ more like $\boldsymbol{b}$ (so state **A** transitions to state **B**). Likewise, when $\boldsymbol{x}$ is most like $\boldsymbol{b}$ it will be changed to be more like $\boldsymbol{c}$ and so on such that the machine can be interpreted to transition in the sequence **ABCABCABC**...

To translate this system into a SPA implementation both the *actions* that the system may take and their *utilities* must be defined. In SPA an action represents the transfer

of information from one module to another and, as in many other cognitive architectures (notably Anderson et al. 2004), each action is associated with a *utility*, a scalar value indicating how beneficial it would be to perform the given action. In our machine there are three actions: "transition to state **B**" (i.e., "assign $\boldsymbol{b}$ to $\boldsymbol{x}$" or $\boldsymbol{x} \leftarrow \boldsymbol{b}$), "transition to state **C**" (i.e., $\boldsymbol{x} \leftarrow \boldsymbol{c}$) and "transition to state **A**". The utility of each of these actions is determined by the current state of the system. For example, the utility of the action "transition to state **B**" is a function of how similar the current state of the system is to the state **A**. This similarity can be measured by computing the dot product between the current state and the vector representing **A**, that is $u(\boldsymbol{x} \leftarrow \boldsymbol{b}) \equiv \boldsymbol{a} \cdot \boldsymbol{x}$. The use of these utilities will be discussed in a few paragraphs but, for now, it is worth noting that the result of undertaking the action $\boldsymbol{x} \leftarrow \boldsymbol{a}$ will be to make $\boldsymbol{a} \cdot \boldsymbol{x} > \boldsymbol{b} \cdot \boldsymbol{x}$ and $\boldsymbol{a} \cdot \boldsymbol{x} > \boldsymbol{c} \cdot \boldsymbol{x}$ – maximising the utility of $\boldsymbol{x} \leftarrow \boldsymbol{b}$. Figure 2.18 illustrates the components that make up this machine and shows the results of a short simulation of the resulting neural implementation.

A 'bus' is one way of thinking about how modules may be combined to form larger, more complex, architectures. For example, many of the modules shown in Figure 2.17 are connected to a bus. By controlling which modules 'write' to and 'read' from the bus different actions can be performed. For example, allowing the module labelled 'Visual Input' to write to the bus at the same time that 'Motor Output' is reading will cause Spaun to attempt to 'draw' what it 'sees'. Spaun will be discussed in more detail below.

In the Semantic Pointer Architecture, this 'bus' is implemented as a model of the *thalamus*, a region of the brain which is implicated in controlling the flow of information to the *cortex* (Sherman 2006). As modelled here, the flow of information through the thalamus is controlled by a model of another brain region, the *basal ganglia*, which is thought to be crucial in making choices between different *actions* (Redgrave, Prescott, and Gurney 1999; Gurney, Prescott, and Redgrave 2001). An action represents, as before, the transfer of information from one module to another: in this case the combination of a 'write' and a 'read' on the 'bus'. The utilities of the various actions are presented to the model of the basal ganglia which selects the action with the greatest utility, causing information to be routed around the architecture. Since the utilities of the actions change as the state of the model evolves, sequences of actions can be performed and the entire system can respond to changes in the external environment (Stewart, Choo, and Eliasmith 2010).

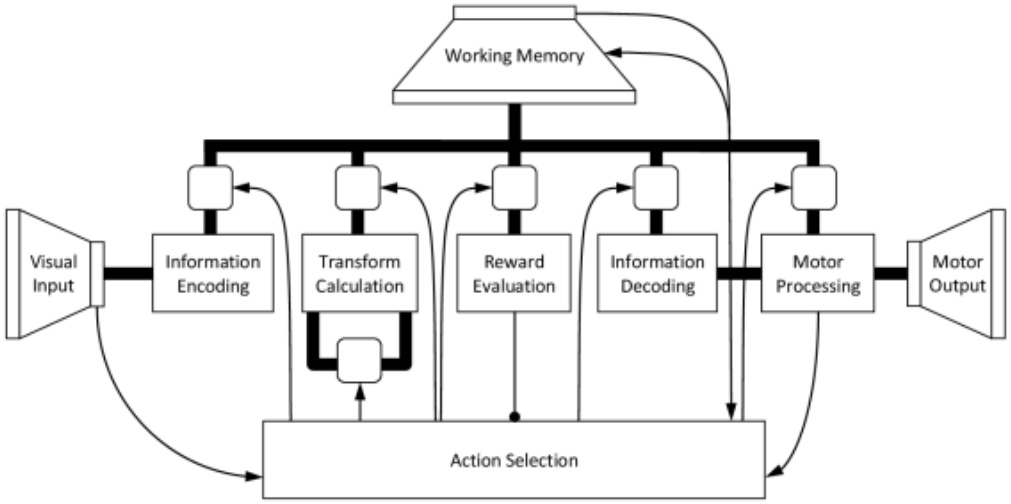


Figure 2.17 – Block diagram of Spaun showing the different modules and their interconnection (Reproduced from Eliasmith 2013). Similar block diagrams could be produced for other cognitive architectures (e.g., Laird, Newell, and Rosenbloom 1987; Anderson et al. 2004). The thick black line represents a *bus* which can be used to transfer values between different modules. Each possible transfer of values across this bus (from 'Working Memory' to 'Motor Output', say) is called an *action* and is associated with a *utility* which indicates how beneficial it would be to perform that action. By repeatedly selecting and performing the action with the greatest utility the overall state of the system can progress.

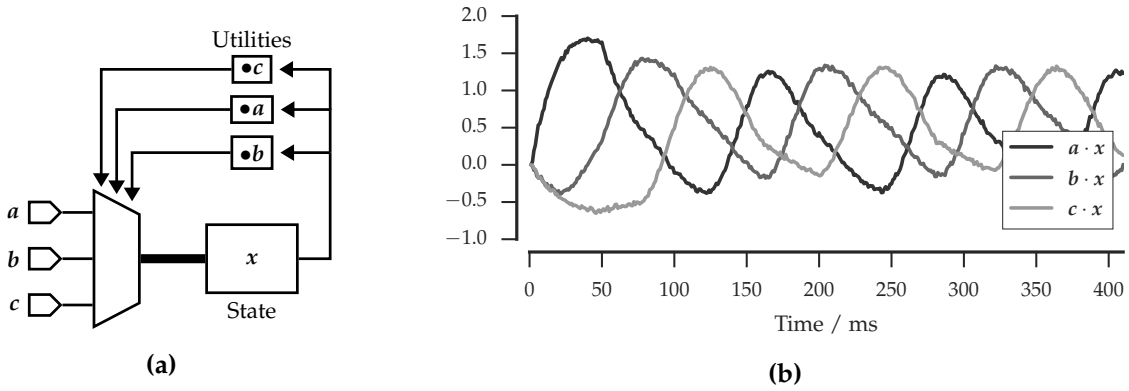


Figure 2.18 – A neural machine. A block diagram of the components is shown in (a). The state of the machine, $\boldsymbol{x}$, is stored in the block labelled "State". The output of this block, also $\boldsymbol{x}$, is compared to each of the three state vectors compute the utilities which are used to select (using the multiplexer [trapezium] in the lower left) the next value of $\boldsymbol{x}$. The time-varying similarity between the state and each of the state vectors is shown in (b). For example, for $0 < t \leq\sim 60$ ms $\boldsymbol{x}$ was most similar to $\boldsymbol{a}$, so the machine can be interpreted as being in state **A** for this period. Likewise for $\sim 60 < t \leq\sim 120$ ms $\boldsymbol{x}$ was most similar to $\boldsymbol{b}$ so for this period the machine can be interpreted as being in state **B**. The neural model contained 2450 neurons and the state was represented with a 16-d vector.

Spaun: The Semantic Pointer Architecture Unified Network model

The Semantic Pointer Architecture Unified Network, Spaun (Eliasmith, Stewart, et al. 2012; Eliasmith 2013), is a neurally-implemented cognitive architecture which ties together the action selection and complex representations of the Semantic Pointer Architecture with the neural representation, transformation and dynamics of the Neural Engineering Framework. Spaun consists of a virtual 'eye', which is fed images representing numbers and letters, a spiking neural implementation of the cognitive architecture that was pictured above and a simulated 'arm' which allows it to 'write' its responses to the inputs that it was given. The model is capable of performing eight cognitive and non-cognitive tasks ranging from reproducing the form of the characters it has been presented to solving problems such as the Towers of Hanoi (Stewart and Eliasmith 2011).

Figure 2.19 shows a sample of Spaun performing a task. In this task Spaun is presented with a series of numbers which form a sequence and must respond with the final element of the sequence. The 'stimulus' line in the figure shows that (after being 'told' which task to perform – A7) Spaun is presented with the sequences: '1, 11, 111', '4, 4, 444' and '5, 55, ?'. The correct response, as shown in the 'arm' line of the figure, is '555'.

2.5 Summary

This chapter has briefly reviewed the basic components from which the human brain is built: neurons and synapses (see Dayan and Abbott 2001, among others, for a thorough review) and two theories which have been used to construct functional models of the brain: the Neural Engineering Framework (Eliasmith and Anderson 2004) and the Semantic Pointer Architecture (Eliasmith 2013). These theories were combined in Spaun (Eliasmith, Stewart, et al. 2012), the "the world's first functional brain model" (Stewart and Eliasmith 2014).

Eliasmith (2013) argues strongly that not only does Spaun represent a significant step toward the goal of tying cognitive behaviour to a neural implementation but also that it reproduces human performance (timing and accuracy) on a series of tasks, including the one shown above. However, given that 1 s of simulation of Spaun required 2.5 h of compute time on a large compute cluster (Stewart and Eliasmith 2014) running sufficient experiments to validate these results is both costly and time consuming. Moreover, since

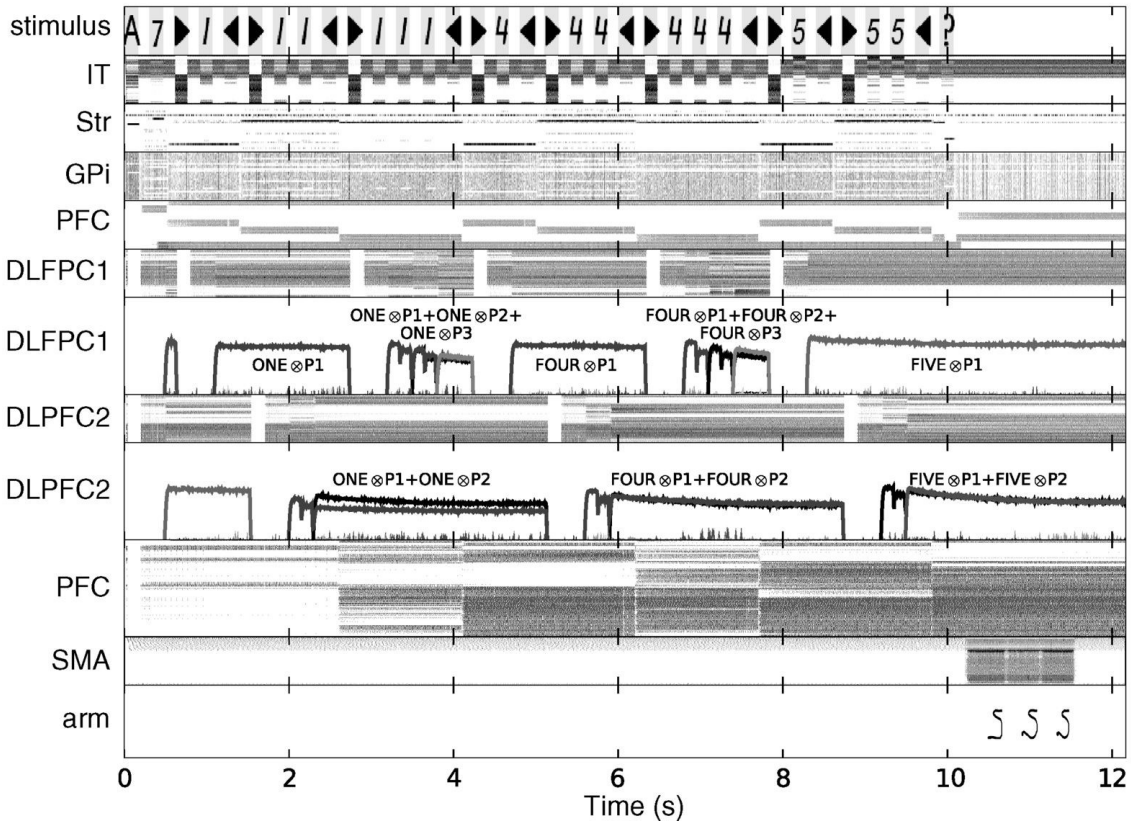


Figure 2.19 – Spaun performing a fluid reasoning task, reproduced from (Eliasmith, Stewart, et al. 2012). The participant of this task is expected to infer the pattern that was used to generate a series of sequences and then to use this rule to complete a sequence. In this case the sequences are '1, 11, 111' and '4, 44, 444' and the pattern is that the character in the sequence is repeated once more every time, such that the response to '5, 55, ?' should be '555'. The 'stimulus' line in the figure shows this input being presented to Spaun and the 'arm' line shows the response.

Spaun is only a fraction of the size of the human brain significant scaling will be required to truly test the hypotheses embodied in the Neural Engineering Framework, Semantic Pointer Architecture and Spaun itself. Consequently, improving the scalability and performance of neural simulators stands to significantly benefit the cognitive sciences.

The following chapter reviews the current state of neural simulation, but before this it is worth considering why *neural* simulation is required at all. For example, the neural machine that was described above is an instantiation of a computational system: its *behaviour*, in the sense of repeated transitions between states, could equally well be described by a finite state machine. However, this fails to capture features of the neural system which may be of interest – for example, how long it takes to transition between states (Stewart and Eliasmith 2009) or how performance degrades if neurons begin to die or a lesion is introduced (Rasmussen and Eliasmith 2014). Consequently, neural models are better placed to investigate the relationship between behaviour and biological implementation than models which, for example, just manipulate semantic pointers. This is highlighted by Eliasmith (2013, p. 219) who shows that the performance of the Spaun model of working memory is at least partially a result of its neural implementation. More generally, a neural model is more descriptive and more easily disproved – features of a good scientific model (see, among others, Popper 1979) – than one which uses SPA alone. Consequently, neural models and simulations fulfil a crucial role in expanding our knowledge of the brain. See Stewart (2012) for more reasoning behind the desire to simulate neurally-implemented cognitive systems.

Chapter 3

Modelling and simulation

The Neural Engineering Framework (NEF) (Eliasmith and Anderson 2004) can be used to construct models which, like Spaun (Eliasmith, Stewart, et al. 2012), tie functional behaviour to a neurally-inspired implementation. By constraining such models by biology, for example by limiting the neurons in a specific component, the hypothesis embodied may be tested through simulation and the measurements compared to those from psychology or biology. However, there are limits to the time and energy available for simulation and this limits the scale and complexity of the models that may be investigated. A number of approaches have been proposed to overcome these constraints.

This chapter first gives an overview of how neural models can be specified using Nengo (Bekolay et al. 2014), a tool for constructing neural systems using the principles of the NEF. In later sections various approaches to large-scale simulations are discussed. Finally, the SpiNNaker architecture, on which this thesis focusses, is introduced.

3.1 Nengo: modelling with the Neural Engineering Framework

Nengo (*N*eural *eng*ineering *o*bjects, Bekolay et al. 2014) is a Python package which allows modellers to specify neural networks using the concepts provided by the Neural Engineering Framework. While Nengo includes a reference simulator implementation, it, like other frameworks such as PyNN (Davison et al. 2008) and Topographica (Bednar 2009), explicitly supports the use of alternative simulators (be they GPUs, supercomputers or neuromorphic hardware). However, unlike either PyNN or Topographica, Nengo is targeted at the construction of neural nets using the principles of the NEF.

The Nengo object model

To model a neural system modellers first construct, using the Nengo data types, a structure which represents the populations of neurons and synaptic connections they wish to investigate (see Sharma, Aubin, and Eliasmith 2016). This structure resembles a graph in which vertices represent populations of neurons (ensembles) and edges the connections between populations. Each ensemble is annotated with information about the parameters and encoders associated with the individual neurons. Meanwhile, the connections between ensembles may either specify a full synaptic weight matrix or a function to be computed by the connection (for example, computing the square of the value represented by the presynaptic population – see Section 2.2). A connection may be labelled with a filter to use to model the filtering which occurs at synapses.

As well as ensembles, Nengo objects exist which allow the recording of simulation data (*Probes*) and allow for the inclusion of stimuli or non-neural phenomena in the model (*Nodes*). A *Node* contains a user-specified function which may receive input from, and transmit output to, the neural network. For example, a Node might be used to simulate a complex component of the network for which no biological model has been proposed, or it may be used to represent data from a sensor, or the values transmitted to an actuator. *Probes* also receive data – such as decoded values, spikes, neuron voltages – from a simulation but, unlike Nodes, they record these data for later inspection.

Nengo simulators

Once the model has been constructed – usually in a manner which is agnostic of the simulation platform – it is passed to a *simulator*. This is responsible for translating the description of the neural network into a form suitable for simulation, running the simulation for a time and returning any results. For example, a simulator targeting a neuromorphic chip might convert the neuron parameters into a form suitable for the device and translate the connections into weight matrices ready to flush into the silicon synapses of the chip. Since the model itself is agnostic of the simulator upon which it will be simulated – save for instances where specific optimisations may be made by the modeller – it may be simulated on any of the *backends* capable of running Nengo models. Currently, such backends include an OpenCL implementation of the NEF and a SpiNNaker backend developed for and partly described within this thesis.

3.2 Simulating neural networks

However a neural model has been specified (whether using, for example, Nengo, PyNN, Topographica or another method) it must be simulated in some form of computational environment. While the differential equation(s) governing the behaviour of a single neuron may be relatively simple, the modelling of many *communicating* neurons is computationally costly. A number of different techniques and architectures have been proposed for efficient simulation of neural nets – some of which eschew the simulation of individual neurons and instead simulate grosser properties of populations of neurons.

The pseudo-code for a simple, clock driven, neural simulator is shown in Figure 3.1. This simulator progresses by computing successive samples of the neuron and synapse states and each 'tick' of the clock drives simulation of the next state of the model. With minor variations this algorithm is the basis of most major neural simulators (e.g., Brian – Goodman and Brette 2009; NEURON – Carnevale and Hines 2006; NEST – Gewaltig and Diesmann 2007).

```
1  t = 0
2  while t < duration
3      for spike in spikes
4          ProcessSpike(spike) // Include spike in inputs of postsynaptic neurons
5      EmptyQueue(spikes)
6      for n in neurons
7          AdvanceNeuron(n)
8          if n.spiked
9              QueueSpike(spikes, n) // Add spike to queue of spikes to process
10     t = t + dt
```

Figure 3.1 – Simple serial neural simulator (adapted from Brette et al. (2007))

The compute time required to simulate the network can be split into that required to simulate the neurons (spent in AdvanceNeuron) and that taken to simulate the spike transmission (spent in ProcessSpike) during which a spike will be distributed to a set of target neurons. Consequently, the total simulation time is a function of not only the size of the network but also the density of the inter-neural connectivity and the firing rates of the neurons. The scale of network which may be simulated this way is limited by how much memory is available to store the network description and state. However,

the simulation of networks which fit within the memory constraint may be exorbitantly expensive in terms of the processing time required.

One way to reduce the wall-clock time required to simulate a large neural model is to use multiple processors and to allocate to each processor a portion of the neurons to be simulated. While the neurons may be simulated entirely in parallel some synchronisation is required to transmit spikes between processors, a message-passing technique for which is described by Morrison et al. (2005). Markram (2006), Izhikevich and Edelman (2008) and Ananthanarayanan et al. (2009) report the use of various supercomputer platforms to perform large scale neural simulations using similar techniques. However, while these authors report significant improvements in the scale and speed of neural simulations which could be performed there remain significant practical limitations.

Overwhelmingly these limitations are related to the costs incurred in synchronising the many processors. For example, Ananthanarayanan et al. (2009) found that of the 173 s compute time required to simulate 1 s of their model 66 s was consumed during synchronisation. The vast majority of this was the result of waiting for all processors to finish work, as some processors had more work to perform than others. Once this was accounted for it was determined that for every 1 s of simulation time 4 s was required to distribute simulation data across the machine. This strongly suggests that real time simulation of neural networks like Spaun on supercomputers is not feasible due to the overheads incurred in synchronisation and communication.

General Purpose Graphics Processing Units (GP-GPUs)

Simulating neurons requires the repeated execution of a small number of operations on a large amount of data. Graphics Processing Units (GPUs) are highly optimised for this form of computation since they consist of many thousands of processors which can apply the same instruction simultaneously to a wide vector of data. Consequently, they can be exploited to perform parallel simulation of neurons. Nageswaran et al. (2009) and Fidjeland and Shanahan (2010), among others, report on large-scale neural simulations using GPUs and highly customised code. However, since making effective use of a GPU requires knowledge of the particular architecture there has been interest in automatically generating code from a high level specification (e.g., Yavuz, Turner, and Nowotny 2016).

While GPUs are undoubtedly useful platforms for the rapid simulation of smaller

neural models there are two keys limits to their scalability. Firstly, GPUs consume significant power (Kindratenko et al. 2009). Secondly, since much of the performance benefit of GPUs derives from the tight coupling of high-performance memory and parallel compute units, accessing data stored in system memory or transferring data between two GPUs or between GPU and CPU causes the performance to drop rapidly (Keckler et al. 2011). For example, Yavuz, Turner, and Nowotny (2016) found that the speed up achieved by a GPU simulation of a neural network, as compared to a CPU, was strongly linked to how much of the 'work' was computational rather than related to memory accesses. In particular, they noted that a ten times greater speed up could be achieved when simulating a model constructed with the mathematically intensive Hodgkin-Huxley neural model than for a model consisting of the considerably less intensive Izhikevich model (see also Pallipuram, Bhuiyan, and Smith 2012). Since larger, or more densely connected, neural networks entail more frequent accesses to memory, both to store and retrieve neural state and to transmit spikes between populations of neurons, the costs of such accesses will likely come to dominate any performance benefits deriving from more parallel computation – as predicted by Amdahl (1967). Consequently, it is fair to state that use of GPUs for neural simulation may be limited by poor scalability.

While near-future GPU and GPU-interconnect architectures are positioned to improve upon this state of affairs by reducing the cost of accessing system memory or transferring data amonst GPUs (Dongarra et al. 2016) they will be subject to the same power constraints as current GPUs. Consequently, GPUs are not well suited to real time simulation of neural models of the scale of Spaun.

Neuromorphic hardware

An alternative approach to parallelising the simulation of neurons is to translate the differential equations that model the neurons into hardware implementations. An extreme instance of this is the use of low power analogue circuits to simulate neurons (Mead 1989; Indiveri et al. 2011); since silicon implementations of neurons are inherently inflexible – once fabricated they cannot be modified – some attempts have been made to construct light weight programmable neurons (Merolla et al. 2014). Although both analogue and digital *neuromorphic* approaches result in rapid, low power, neural simulation they are subject to some severe limitations.

While simulating neurons can be computationally costly (particularly for models such as Hodgkin-Huxley) it is the communication between the neurons which dominates the overall cost of the simulation. Therefore, although the massive parallelism that can be achieved with low-level neural model implementations does reduce the time and power consumed it does not eradicate the problems associated with transmitting spikes or simulating synapses. Since a combinatorial "explosion" would occur if synapses were constructed between every pair of neurons most large scale neuromorphic systems instead use a digital network to transmit spikes and provide each neuron with a fixed number of incoming synapses (Schemmel et al. 2010; Choudhary et al. 2012; Merolla et al. 2014). This limits the number of connections that can be made to a neuron. While a neuron can often 'borrow' synapses from another this comes at the cost of reducing the number of connections that can be made to the other neuron. Consequently, neural networks, like those constructed with the Neural Engineering Framework, which feature dense inter-neural connectivity can be expected to make very poor use of current neuromorphic platforms.

Field-Programmable Gate Arrays (FPGAs)

Since a significant disadvantage of many neuromorphic platforms was their inflexibility Field Programmable Gate Arrays (FPGAs) have been suggested as an alternative (Cox and Blanz 1992). An FPGA, which contains a number of programmable logic blocks and memory elements which may be connected together to build new computational elements, can be used to allow relatively cost effective and rapid development of custom hardware. For example, Berzish, Eliasmith, and Tripp (2016) demonstrate a population-based implementation of the Neural Engineering Framework, capable of simulating networks of around one million neurons using a single chip. In this particular instance, a surrogate model of the populations (Tripp 2015) was used to allow simulation with a longer time step and reduce the number of parameters associated with each population of neurons, subsequently reducing the comparatively expensive memory accesses. Cassidy, Andreou, and Georgiou (2011) and Moore et al. (2012), among others, report on more general neural simulation using FPGAs.

3.3 SpiNNaker

Previous approaches to neural simulation, aside from the neuromorphic systems, have been composed of relatively few large and expensive compute elements. Since components were costly, increasing the scale of simulations required increasing the time required for their execution rather than increasing the number of compute elements that were used. In contrast, the SpiNNaker architecture (Furber, Galluppi, et al. 2014; Furber and Temple 2007) – designed specifically for the simulation of spiking neural networks – is constructed of many low-power and cheap compute elements, making for a highly scalable architecture. Furthermore, since one of the largest costs in supercomputer simulation of neural networks was synchronisation (Ananthanarayanan et al. 2009) SpiNNaker runs *asynchronously*.

Hardware

A SpiNNaker *machine* is constructed from a number of SpiNNaker chips. Each chip contains 18 low-power processing cores, each with 32 KiB of instruction memory (ITCM) and 64 KiB data memory (DTCM). In addition to these small amounts of private memory is a 128 MiB SDRAM which is shared amongst all the processing cores on a chip. Since reads and writes to this shared memory are subject to significant latency cores may issue asynchronous Direct Memory Access (DMA) requests, allowing programs to continue to perform useful work while memory access is performed.

SpiNNaker cores execute small kernels which communicate amongst themselves by passing short messages across a communications network-on-chip (NoC). To allow messages to be sent between the kernels being executed by cores on different chips the communications NoC is connected to an inter-chip network. This network links each chip with six neighbours and the entire fabric may be wrapped into a torus to maximise the aggregate bandwidth of the machine. The links connecting a node to its neighbours are commonly labelled with compass points. Packets are directed around this network by *routers* contained within each SpiNNaker chip. The components of a SpiNNaker chip and the arrangement of chips to form a mesh network are shown in Figure 3.2.

The messages passed amongst processing cores consist of an 8 bit header, a 32 bit key and an optional 32 bit payload. Once transmitted by a processing core these packets travel through the network by traversing the inter-chip links. At each chip the packet is

inspected by a router which may duplicate and forward the packet to any combination of the 18 processing cores located on the chip and six inter-chip links. Routers determine how a packet should be routed by comparing its key against a 1024 entry *routing table,* implemented as a Ternary Content Addressable Memory (TCAM). Figure 3.3 illustrates the tree of routes that might be taken by a single packet when it is multicast across the SpiNNaker network. Routers and routing tables are described in more detail in Section 6.1.

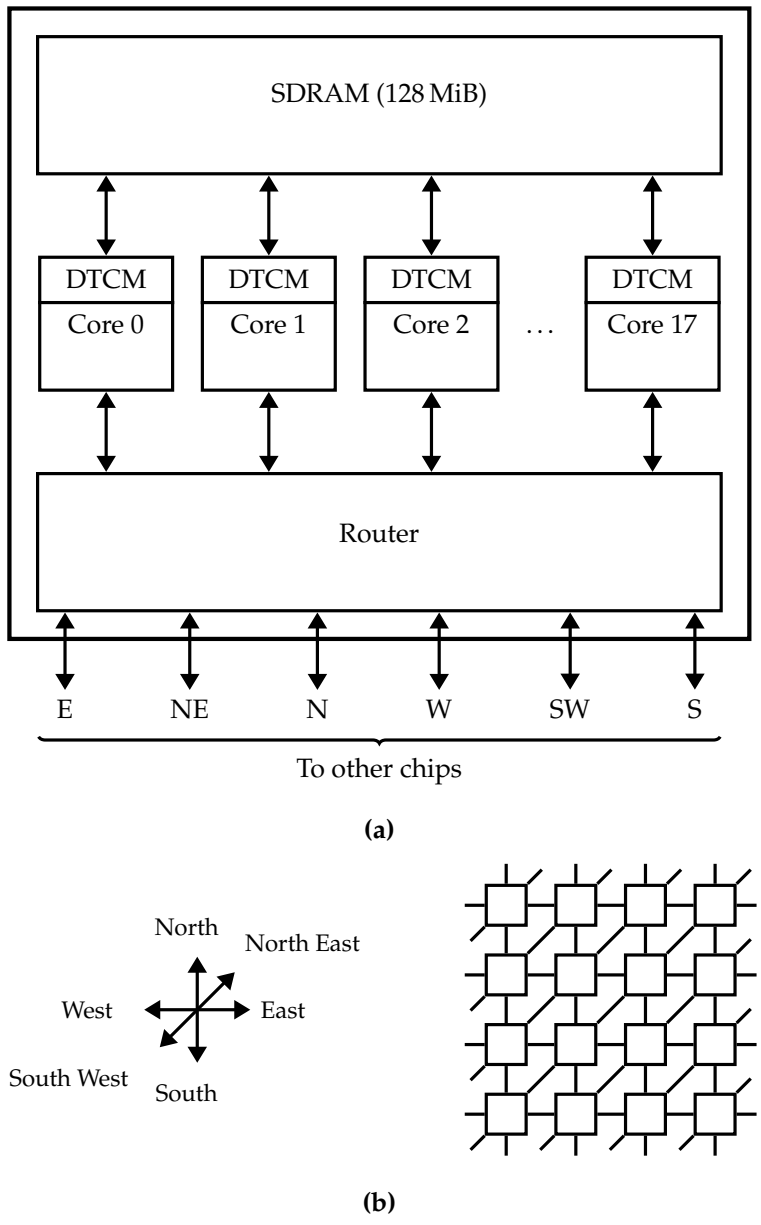


Figure 3.2 – The components of a SpiNNaker chip are illustrated in (a). Many chips can be combined to form a mesh network as shown in (b).

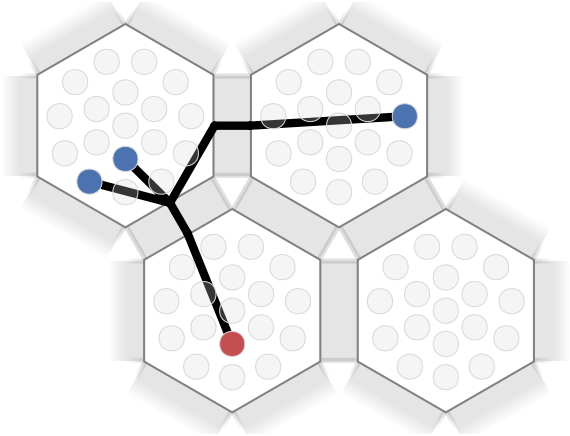


Figure 3.3 – Multicasting packets across the SpiNNaker architecture. Each hexagon represents a SpiNNaker chip and each circle a SpiNNaker core. The tree drawn across the SpiNNaker machine shows the route that might be taken by a packet transmitted by the red core and destined for delivery to the blue cores; at the fork the packet is duplicated.

There are a number of limitations to the SpiNNaker architecture. To save energy and area no floating point hardware is provided and, because software implementations of such operations are not performant, fixed point representations are used instead. Hopkins and Furber (2015) discuss the implications this has for neural simulation and Section 4.5 demonstrates some of these effects with respect to the Neural Engineering Framework (NEF).

Additionally, it is not guaranteed that all SpiNNaker packets will be delivered to their intended recipients: packets can be *dropped* to reduce congestion if the network is overloaded. For the spiking neural networks for which the architecture was designed it was assumed that, since neurons are inherently noisy and unreliable, occasional dropped packets would not have a significant effect upon simulation results.

Neural simulation

When simulating a spiking neural network on SpiNNaker each core is assigned a number (typically of the order of a few hundred) of neurons to simulate. Figure 3.4 illustrates how the populations of a neural network might be mapped to a SpiNNaker machine.

SpiNNaker cores are programmed in an event driven manner and are triggered by events such as the ticking of a timer, the receipt of a multicast packet and the completion of a read or write to the shared SDRAM present on each chip. The serial neural network simulator of Figure 3.1 may be mapped to this event driven system. First, a timer

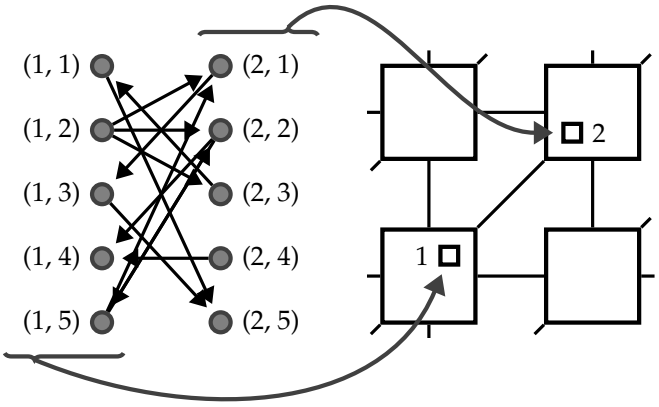


Figure 3.4 – Sample neural network mapped to a SpiNNaker machine. The neural network consists of two populations, with one population mapped to a core on the lower-left chip and the other mapped to a core on the upper-right chip.

is configured to 'tick' at a regular interval; usually with a period chosen to be the same as the time step of the simulation such that the simulation progresses in real time. The timer 'tick' triggers the update of the synapse filters and neuron models assigned to the processing core. If, during this update, a neuron fires then a multicast packet is transmitted. The key of this 'spike-packet' is assigned a value which uniquely identifies the firing neuron (see Boahen et al. 2000, for more on Address Event Representation (AER)). Once the state of all neurons has been updated the core idles until triggered by the next event.

Meanwhile, the spike packets traverse the SpiNNaker network and are multicast to all the processing cores which simulate neurons connected to the source neuron. The receipt of a packet causes a core to read the row of the synaptic weight matrix associated with the firing neuron. The synaptic weights contained in this row are then added to the inputs for all of the postsynaptic neurons and these inputs will be used to update the neural states upon the next tick of the timer. The entire process is illustrated by Figure 3.5.

There are limits on the number of neurons which may be simulated by a core. Firstly, there must be sufficient memory available to store the neural parameters and state. Secondly, it must be possible to store the synaptic weight matrices in SDRAM. Since the weight matrices are assumed to be sparse they are stored in a compressed form to save memory. Unfortunately, the weight matrices of neural networks constructed using the Neural Engineering Framework are *not* sparse (Section 2.2) and this has ramifications for how they are simulated (Section 4.1).

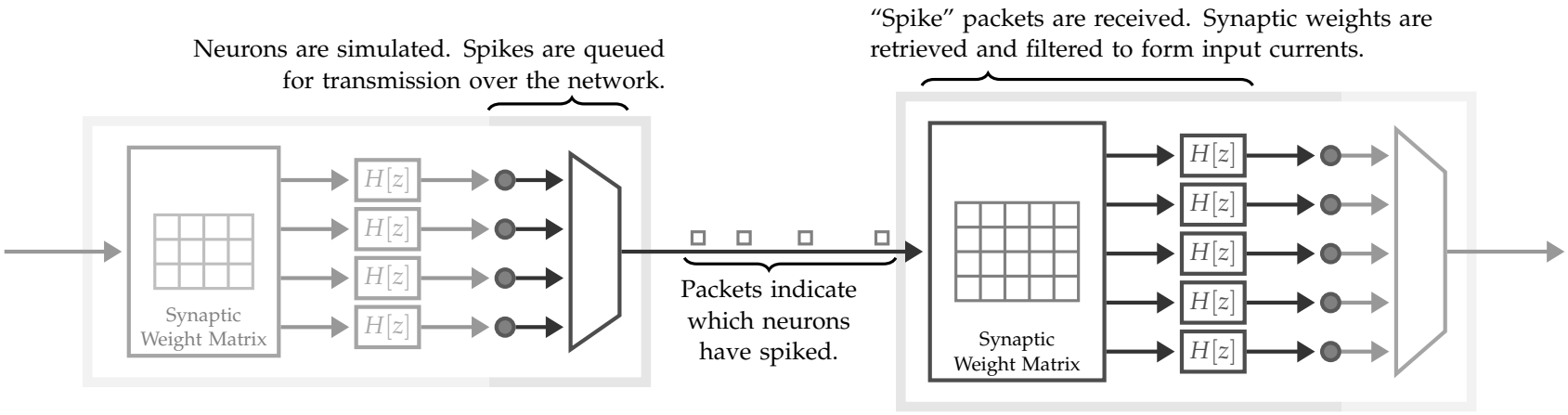


Figure 3.5 – Simulation of neural nets on SpiNNaker

Since each processing core must have completed all the tasks associated with a step of the simulation before the next tick of the timer there is also a *computational* constraint on the number of neurons which can be allocated to a single core. The computational load on a single core is a combination of the time taken to simulate the neurons and that required to process the synaptic rows. Sharp and Furber (2013) found that the time required to process incoming spikes was one of the biggest factors in SpiNNaker performance. Every received spike results in the retrieval of a row of the synaptic weight matrix from SDRAM and every value in the retrieved row must be included in the input for a neuron. Each of these values constitutes a *synaptic event*. Sharp and Furber (2013) found that a processing core could only process 5000 synaptic events per millisecond before missing its real time deadlines.

The constraints on processor utilisation and memory usage may be satisfied by allocating fewer neurons to each processor core. This reduces the time that will be spent updating neural states, the amount of DTCM required to store the neural parameters and state, and the length of the rows of the synaptic weight matrices. Reducing the length of the rows reduces the number of synaptic events which will need to be processed – which, in turn, reduces the computational load – and the memory required to store the matrices. However, shorter rows mean that the fixed costs associated with receiving spikes and retrieving synaptic rows from memory can be amortised over fewer synaptic events – essentially increasing the cost of each event (Knight and Furber 2016).

Placement and routing

The previous section described the steps taken by kernels running on SpiNNaker cores during the simulation of a neural net. However, before this stage of the simulation can begin there are several steps necessary to convert the abstract representation of the network produced by Nengo, PyNN or similar into a form which can be simulated on SpiNNaker and to load applications and data structures to the machine. These pre-processing steps can be divided into *partitioning*, *placing and allocation*, *routing* and *loading*.

The description of neural networks produced by Nengo can be thought of as a graph whose vertices are representative of computational work (e.g., ensembles to simulate or values to record) and whose edges represent communication between these computational elements. Figure 3.6 shows this correspondence between a neural network and a

less detailed graph-like view. In this example there is a connection between two populations of neurons, since the graph merely represents that there *is* some communication between the ensembles it does not need to show the exact weight matrix used.

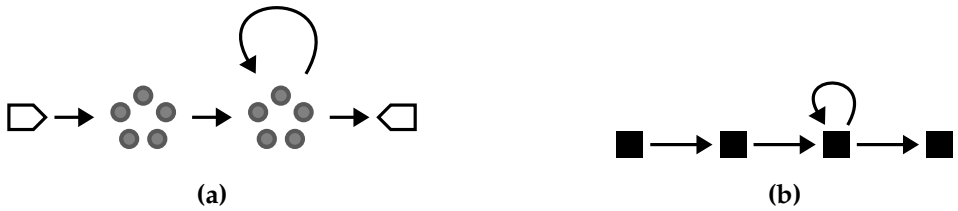


Figure 3.6 – Neural networks can be abstracted as data flow graphs for the purpose of preparing them for simulation on the SpiNNaker platform. The neural network as a graph-like structure is shown in (a) and the data flow graph is shown in (b).

Some of the units of computational work in this graph might be larger than can be accommodated on a single SpiNNaker core and these must be *partitioned*. For example, a population of neurons which requires the processing of too many synaptic events must be split to bring the load of each of the resulting partitions below the constraints imposed by a single core. Figure 3.7 shows how this would proceed. The population is split into as many partitions as required to meet the compute and memory constraints and the connections to and from the ensemble are duplicated to show the new connectivity of the network. After partitioning, the vertices of the graph represent units of computational work that will be assigned to SpiNNaker cores and the edges correspond to the flows of packets between processing cores required to simulate the neural network.

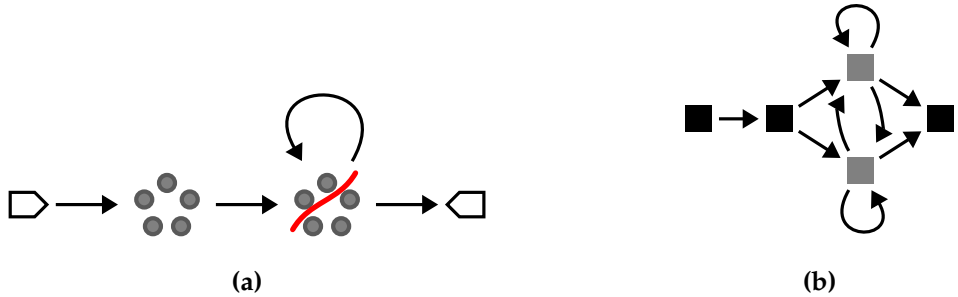


Figure 3.7 – Partitioning a population of neurons to meet compute and memory constraints. The neural network is shown as a graph-like structure in (a) and the ensemble to be partitioned is highlighted. The data flow graph, with the ensemble partitioned, is shown in (b).

One could proceed to assign each of these units of computational work (vertices in the data flow graph) to a randomly selected core in the machine and generate the routing tables to ensure that multicast packets were routed as required, as illustrated in Figure 3.8. However, each of the inter-chip network links has a limited bandwidth and this suggests that the *placement* of vertices should be performed in such a way that densely connected vertices are placed more closely than loosely connected vertices and that multicast packets be *routed* to minimise network congestion. A number of schemes for achieving this are described by Heathcote (2016).

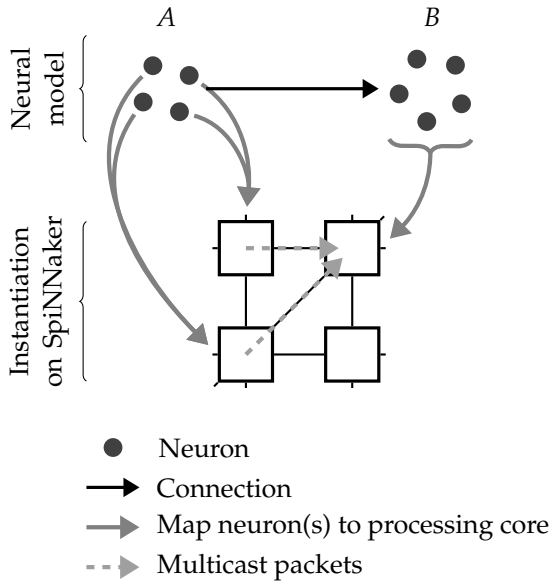


Figure 3.8 – Partitioning, place and routing a neural network.

The result of the *placing and routing* the network is that each ensemble, Node and Probe in the original Nengo network is mapped to a SpiNNaker core and that routing tables exist which ensure that the streams of multicast packets used to simulate connections between network objects are routed correctly. Finally, before simulation can begin, simulation data, routing tables and application executables are loaded to the SpiNNaker machine. Ensuring that placing, routing and loading occur as fast as possible is a key aim of the SpiNNaker project, not only to reduce the wall-clock time required to simulate a neural network but also because Diamond, Nowotny, and Schmuker (2015) found that the majority of energy consumed when using SpiNNaker was consumed during these stages when the SpiNNaker machine was largely idle.

Chapter 4

The Neural Engineering Framework and SpiNNaker

Section 4.1 analyses how the Neural Engineering Framework (NEF) might be mapped onto the SpiNNaker architecture and highlights the problems that would occur if the neural simulation technique that was described in Section 3.3 were used. Following this analysis, Section 4.2 proposes a novel implementation of the NEF which transmits neuronal activities between processors as *values* rather than as the *spikes* detailed in previous sections – this reduces the memory, compute and network resources required to simulate neural networks built with the NEF. These two sections (4.1 and 4.2) reproduce work first presented in "An efficient SpiNNaker implementation of the Neural Engineering Framework" (Mundy, Knight, et al. 2015). Section 4.3 extends this novel implementation of the NEF to reduce the network traffic required to simulate neural models. Finally, Sections 4.4 and 4.5 present results pertaining to both the performance and correctness of the novel NEF implementation for SpiNNaker.

4.1 Mapping the Neural Engineering Framework to SpiNNaker

Under the "spike-transmission" scheme for simulating neural networks (Section 3.3) there are two major constraints on how many neurons may be allocated to each core:

1. The synaptic weight matrix must fit within the memory available to the processor. Nominally an 8 MiB share of the 128 MiB SDRAM per-chip is allocated to each core.
2. As the majority of the processing time is spent in the synaptic processing pipeline

(Sharp and Furber 2013), there must be sufficient time for the core to receive all "spike" packets and retrieve and process the synaptic rows during one simulation time step. The processing time required is a function of both the number of incoming spikes per time step and the density of the synaptic weight matrix. Sharp and Furber (2013) indicate that there may be at most 5000 synaptic events per time step, where a single synaptic event indicates a single spike being passed through one synapse to one neuron on the receiving core.

These constraints may be satisfied by either allocating fewer neurons to each processing core (reducing the size of the weight matrix and thus the number of synaptic events), or by allowing more compute time per simulation time step (increasing the number of synaptic events that may be handled). However, allocating fewer neurons to a processing core increases the number of cores necessary to perform a particular simulation and may mean that a given simulation becomes infeasible given the hardware available. The alternative, running SpiNNaker slower than biological real time, is undesirable for a real time simulator and may not be possible if the simulation is intended to interface with a neuromorphic sensor or actuator.

Neural networks constructed using the principles of the NEF have two characteristics which pose significant challenges to the SpiNNaker architecture:

1. Weight matrices under the NEF tend, unlike those commonly found in SpiNNaker simulations, to be dense. Consequently more memory is required for their storage; and, as each row contains many values, many more synaptic events are associated with each incoming spike.

2. Neurons in NEF simulations tend to have significantly greater firing rates than the SpiNNaker platform was designed for. This increases not only the network traffic, but also the number of synaptic events processed by each active core.

Example

Consider two connected 50-neuron ensembles, each representing a scalar. Assuming dense matrices and using 16 bit values for the synaptic weights, the postsynaptic core must store $50 \times 50 \times 2\,\mathrm{B} = 5000\,\mathrm{B}$. Since this is less than 8 MiB it can be stored with no problem. With a row-length of 50, as many as 100 incoming spikes per millisecond

can be received before reaching the limit of 5000 synaptic events per time step. As this is double what could be produced by the 50-neuron presynaptic ensemble the computation constraint is satisfied. Consequently, this trivial NEF network can be simulated on SpiNNaker without splitting either population across cores.

Now consider two connected ensembles representing a d-dimensional space. For an ensemble to represent a multidimensional value accurately it must contain sufficient neurons – between $50d$ and $70d$ (Eliasmith 2013). The weight matrix between two such ensembles consumes $50^2d^2 \times 2\,\mathrm{B} = 5000d^2$ B and will outstrip the available memory as d scales. The computational load also grows with d. Figure 4.1 shows how the number of synaptic events grows with the increasing number of neurons in the ensembles; beyond 200 neurons the mean compute load is beyond that found by Sharp and Furber (2013) to be sustainable. For an 800 neuron ensemble, a common size in Spaun, a maximum of 57 neurons may be allocated to a processing core before the compute load associated with processing synaptic events becomes untenable – an order of magnitude below the architectural target of a thousand neurons per core (Furber and Temple 2007).

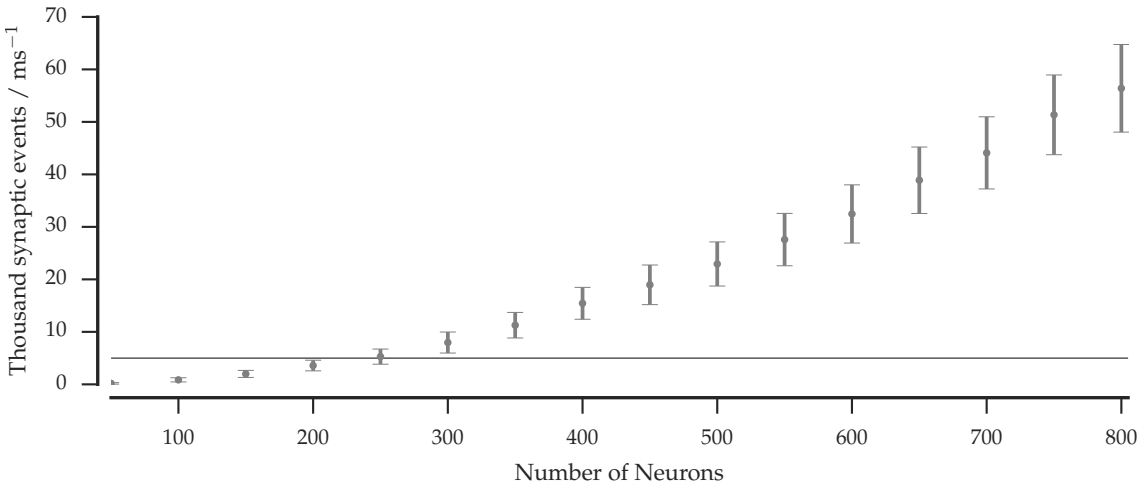


Figure 4.1 – Number of synaptic events occurring in the postsynaptic ensemble of a pair of n-dimensional ensembles.

4.2 Communicating with values, not spikes

Section 4.1 highlighted that the dense weight matrices and high firing rates of the Neural Engineering Framework would result in poor use of the architecture if mapped directly to SpiNNaker. In this section an alternative simulation scheme is described, which leads to much improved use of the platform.

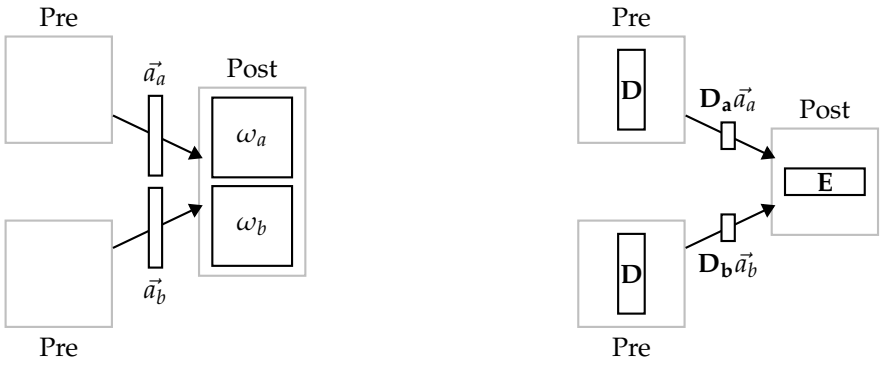
In Section 2.2 we saw that, under the NEF, the synaptic weight matrix between two populations could be computed through multiplication of the *encoders* of the postsynaptic population with a *decoder* of the presynaptic population: $\omega = \mathbf{ED}$. If the output of the presynaptic neurons is thought of as a vector whose elements are either 0 or 1, to indicate "not spiked" or "spiked" respectively, then the input to the postsynaptic neurons is formed by premultiplying the spike vector by the weight matrix. In the standard, spike-based, SpiNNaker simulation technique this matrix-vector multiplication occurs implicitly: each received spike triggers the retrieval of a row of ω and adds the non-zero elements of that row to the inputs for the postsynaptic neurons. However, as discussed above, storing dense-ω for networks built with the NEF is expensive and, as firing rates can be expected to be high, transmitting the spike-vector across the network is also costly.

Since the matrix $\omega = \mathbf{ED}$ is known to be exactly factorisable an alternative implementation of the NEF for SpiNNaker is suggested, wherein we perform two smaller matrix-vector multiplications explicitly. The standard SpiNNaker simulation essentially computes $\boldsymbol{J} = \omega \boldsymbol{a}$, where $\boldsymbol{J}$ is the input to the postsynaptic neurons and $\boldsymbol{a}$ is the spike-vector from the presynaptic neurons. Since $\omega = \mathbf{ED}$, and $\mathbf{ED}\boldsymbol{a} \equiv \mathbf{E}(\mathbf{D}\boldsymbol{a})$ we may perform the operation $\boldsymbol{x} = \mathbf{D}\boldsymbol{a}$ on the core simulating the presynaptic neurons and the operation $\mathbf{E}\boldsymbol{x}$ on the core simulating the postsynaptic neurons. Figure 4.2 illustrates the changes to data-flow and data-storage required to make this change.

(a) Non-factored weight matrix **(b)** Factored weight matrix

Figure 4.2 – Comparison between non-factored and factored weight matrices for a single connection. Note that the factored weight matrices are split between the pre- and postsynaptic cores and that the product $\mathbf{D}\boldsymbol{a}$ rather than the vector $\boldsymbol{a}$ is transmitted between the cores.

There are several benefits to this scheme. Firstly, whereas the matrix **ED** was of the shape $N_{\text{post}} \times N_{\text{pre}}$, the factored matrices are considerably smaller and may be stored locally within each core. Consequently, where the postsynaptic core previously needed to store **ED** it now need only store **E**, which is of shape $N_{\text{post}} \times d$, however where the presynaptic core did not need to store anything with regard to the connection it must now store **D** of shape $d \times N_{\text{pre}}$. As $d \ll N_{\text{pre}}$ and N_{post} an overall memory saving is achieved. Furthermore, as $\mathbf{E_j}$ is a factor of all the connection weights arriving at population j it need not be duplicated for multiple incoming connections (see Figure 4.3), consequently multiple incoming connections require no more memory from the postsynaptic core than is required for one connection.



(a) Non-factored weight matrices **(b)** Factored weight matrices

Figure 4.3 – Comparison between non-factored and factored weight matrices for a fan-in of two connections. Since **E** is a factor of both ω_a and ω_b it need not be duplicated; reducing both the amount of computation required and the memory necessary.

However, where it was previously necessary to transmit the vector $\boldsymbol{a}$ between processing cores the new scheme requires transmission of $\boldsymbol{x}$ $(= \mathbf{D}\boldsymbol{a})$. In the spike-based simulation scheme, the vector $\boldsymbol{a}$, elements of which indicate the presence or absence of a spike, was transmitted between cores by sending (or not sending) a multicast packet for each element. This technique is not appropriate for transmission of $\boldsymbol{x}$, the elements of which are numbers, not boolean values. An appropriate coding is to transmit a single multicast packet for each of the d elements of $\boldsymbol{x}$ with the key indicating the index of the element $(1 \ldots d)$ and the payload containing the value $((\mathbf{D}\boldsymbol{a})_{1\ldots d})$. As a result, the presynaptic core can be expected to transmit exactly d packets per time step whereas under the spike-based scheme a packet would have been sent for each firing neuron. Since d, the number of dimensions represented by an ensemble, is typically much less than n, the

number of neurons in the ensemble, the result is a significant reduction in the number of network packets required to model communication between neural populations.

Example

The value-based simulation scheme will be described in greater detail later. For now, let us return to the example of two connected ensembles, each representing an d-dimensional space and containing $50d$ neurons.

Under the spike-based scheme we identified that $(50d)^2 \times 2\,\mathrm{B} = 5000d^2$ B was required to represent the synaptic weight matrix resulting from the connection between the two ensembles. When using the value-based scheme we store the weight matrix as two factors: **E** and **D**. Each of these factors is of size $50d \times d$; using 4 B per element results in $200d^2$ B being stored by both the pre- and postsynaptic processing cores. For $d = 16$, a common size in Spaun, the full synaptic weight matrix would require 1250 KiB whereas the factored weight matrices consume only 100 KiB – a saving of 92 %.

However, it is not only in memory usage that savings are made through use of factored weight matrices. For $d = 1$, the value-based scheme will result in the transmission of one long (with payload) multicast packet per simulation time step – or a throughput of $72\,\mathrm{kbit\,s^{-1}}$. In contrast, we can expect the presynaptic core to transmit between zero and fifty short (without a payload) multicast packets per simulation time step – depending on the firing rates of the neurons, and assuming we follow the $50d$-neurons heuristic – a throughput of up to $2000\,\mathrm{kbit\,s^{-1}}$. Figure 4.4 illustrates how the throughput between the cores varies with the value represented by the presynaptic core. In this example we see that the communication bandwidth varies from $100\,\mathrm{kbit\,s^{-1}}$ to $400\,\mathrm{kbit\,s^{-1}}$ for the spike-based scheme but is a constant $72\,\mathrm{kbit\,s^{-1}}$ for the value-based scheme.

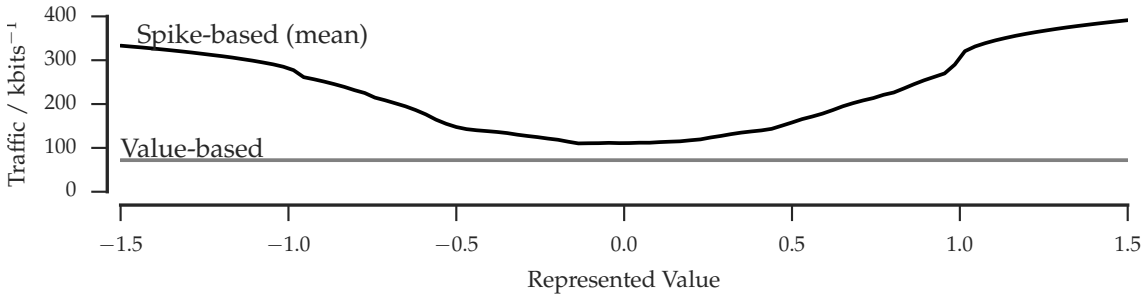


Figure 4.4 – Sample traffic rates between two 50-neuron ensembles each representing a 1-dimensional value.

Simulating neurons

Under the value-based method of simulating neural nets on SpiNNaker each core is, as before, allocated a number of point neurons to simulate. Each simulation step is split into three phases: *input filtering*, *neuron update* and *output*; which correspond to simulating synaptic filters, neurons and spikes respectively. The stages are illustrated in Figure 4.5, beginning with the *neuron update*.

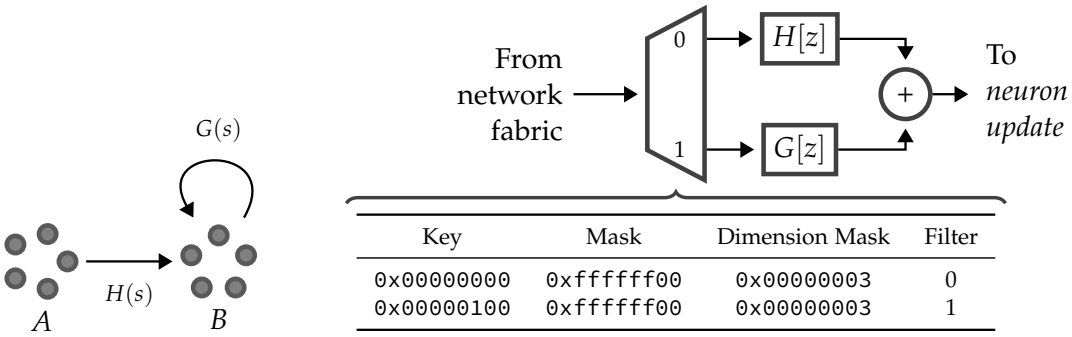
During the *neuron update* phase each neuron is simulated, with the input to the neuron formed from the dot product of the encoder for the neuron (a row in $\mathbf{E}$) and the input vector resulting from the *input filtering step*. Should a neuron *fire* then the row of the decoder matrix ($\mathbf{D}$) associated with it is added to vector representing the output of the neurons for this time step. Once all neurons have been simulated the *output phase* begins, during which multicast packets are used to transmit the vector formed during the *neuron update* stage. Since each packet can carry at most a single 32 bit payload, multiple packets are required to transmit a vector composed of 32 bit elements. To allow reconstruction of the vector at the receiver the packet key is used to indicate the index of the vector element as well as to route the packet across the network fabric to receiving processing cores. This use of the key exploits the SpiNNaker network model, which allows bits of the key to be masked off for routing purposes while allowing the remainder to carry information unrelated to routing.

Simulating synapses

Once a packet has been received by a processing core it must be included in the input for a synaptic filter. Since an ensemble may receive multiple inputs which must be filtered with different synapse models, each received packet is routed to a filter by comparing it against a lookup table.

A neural net and lookup table are shown in Figure 4.6. This net contains two connections which terminate at ensemble B: one connection, labelled with the filter $G(s)$, is recurrent and the other connection, labelled $H(s)$, originates at A. Packets from A and B form two vectors, $\boldsymbol{a}$ and $\boldsymbol{b}$, which must be filtered separately to produce the input for B, $H(s)\boldsymbol{a} + G(s)\boldsymbol{b}$. The synaptic filters, $G(s)$ and $H(s)$, are instantiated on the core simulating B as $G[z]$ and $H[z]$ respectively and a lookup table is used to include the payloads of the packets with the correct filter.

Figure 4.5 – "Value"-based method of simulating neural nets on SpiNNaker



Key	Mask	Dimension Mask	Filter
0x00000000	0xffffff00	0x00000003	0
0x00000100	0xffffff00	0x00000003	1

(a) Neural model

(b) Packet to filter routing on SpiNNaker

Figure 4.6 – An example of how different synapse models can be instantiated on SpiNNaker, (a) shows a simple neural model with two neural connections terminating at population B. The structures necessary to route packets, once received, to their appropriate synapse models are illustrated in (b).

A sample lookup table is shown in Figure 4.6(b). In this example, the processing cores simulating A transmit multicast packets with keys which begin `0x000000XX` and those simulating B transmit packets with keys beginning `0x000001XX`. Therefore, a packet arriving from A with the key `0x00000021` would match against the first entry (since `0x00000021 & 0xffffff00 = 0x00000000` – the packet key, when masked, is equal to the key in the entry) and would be included in the input vector for filter $H[z]$. To determine with which element of the input vector for $H[z]$ the payload should be included the key is masked with the 'Dimension Mask' included in the entry. In our example, as `0x00000021 & 0x00000003 = 0x1` the payload would be added to the second element of the input vector for $H[z]$. A similar process would occur for packets arriving from cores simulating B except that these packets would match the second entry in the table.

The compute cost of comparing a packet against the lookup table is a function of the number of entries overall and the number of entries which match the packet. Reducing the compute cost may be necessary to ensure that packets are removed from the network fabric fast enough, or to avoid buffers overflowing, and this may be achieved by using logic minimisation to represent the table more compactly. Chapter 6 investigates various techniques for minimising both network routing tables and these synapse routing tables.

In the *input filtering* step the synaptic filters are updated using the input vectors formed from packets received during the previous time step. Once updated, the outputs of the filters are combined to produce a single input for the ensemble and this vector is used to form inputs for the next set of neurons in the *neuron update* phase.

4.3 Using shared-memory parallelism to reduce network traffic

The previous section described the value-based simulation scheme as originally presented by the author (Mundy, Knight, et al. 2015). This simulation scheme was found to support up to 2400 neurons per core when 1-D representations were used, more than twice the SpiNNaker architectural target and, as described, resulted in significantly lower memory usage than the spike-based scheme. However, memory consumption and compute requirements may still entail the splitting of populations of neurons across cores, leading to a significant drawback of the scheme.

Consider two connected 800-neuron ensembles, each representing a 16-D space. The encoders for each population consume $800 \times 16 \times 4\,\mathrm{B} = 50\,\mathrm{KiB}$, or most of the DTCM available to a core. To ensure sufficient room is left for other data structures (decoders, neural parameters, states etc.) each population is spread across three processors. Every time step the three presynaptic cores each simulate around 260 neurons and construct a 16-D decoding of their output, to be transmitted as 16 multicast packets. Since there are three presynaptic cores, each transmitting 16 packets, the postsynaptic cores receive 48 packets per time step. Under this value-based simulation scheme partitioning an ensemble p-ways leads to a factor p increase in the number of packets received by each subsequent processing core – an undesirable trend. In comparison, under the spike-based scheme, partitioning populations of neurons across multiple cores does not lead to an increase in the overall network traffic and may, in certain circumstances, have the desirable effect of spreading the traffic more thinly across more of the network.

The value-based simulation scheme can, as well as increasing the total network traffic when partitioning populations, suffer from heavy traffic bursts. In the scheme as described above, neurons are immediately decoded during the *neuron update* phase to contribute to a vector transmitted during the *output* phase. As transmitting an element of the output vector is relatively fast not much time separates the multicast packets as they flow through the network and this significantly increases the likelihood of packets being dropped. In the value-based simulation scheme described in our paper (Mundy, Knight, et al. 2015) a fixed delay was inserted between each packet; this, however, wastes processor cycles and still results in bursty network traffic. In the spike-based scheme, by comparison, packets are separated by at least the time taken for one neuron to be simulated – meaning that cycles need not be wasted to ensure packet separation.

This section extends the value-based simulation scheme to reduce the overall network traffic and to decrease the burstiness of said traffic without inserting arbitrary delays between the transmission of multicast packets.

Overview of the solution

Thus far SpiNNaker has been used solely as a message-passing parallel computer. However, the shared SDRAM could be used to transfer data amongst cores located on the same chip. There are two places in the value-based scheme where performance can be improved through use of this shared memory: decoding and synaptic filtering.

The decoding of an ensemble, when partitioned across two cores, is illustrated in Figure 4.7(a). Since the spike vector, $\boldsymbol{a}$ of size n is split into two parts, $\boldsymbol{a}_a$ and $\boldsymbol{a}_b$, of size $\frac{n}{2}$ the decoding matrix, $\mathbf{D}$ of size $d \times n$ must be split column-wise into two parts, $\mathbf{D}_a$ and $\mathbf{D}_b$, of size $d \times \frac{n}{2}$. The resulting matrix-vector products, $\mathbf{D}_a\boldsymbol{a}_a$ and $\mathbf{D}_b\boldsymbol{a}_b$, are of size d and must be added to give the result $\mathbf{D}\boldsymbol{a}$ – the decoding of the entire population. Figure 4.7(b) shows how the partial products – $\mathbf{D}_a\boldsymbol{a}_a$ and $\mathbf{D}_b\boldsymbol{a}_b$ – may be computed on different processing cores, transmitted across the network and summed by the receiver to form the product $\mathbf{D}\boldsymbol{a}$. Again we note that a partitioning of the problem has doubled the network traffic, since each transmitting core transmits d packets despite having only simulated one half of the neurons (constructed one half of $\boldsymbol{a}$). Moreover, the vector must be summed at every receiving processing core rather than being summed at the source.

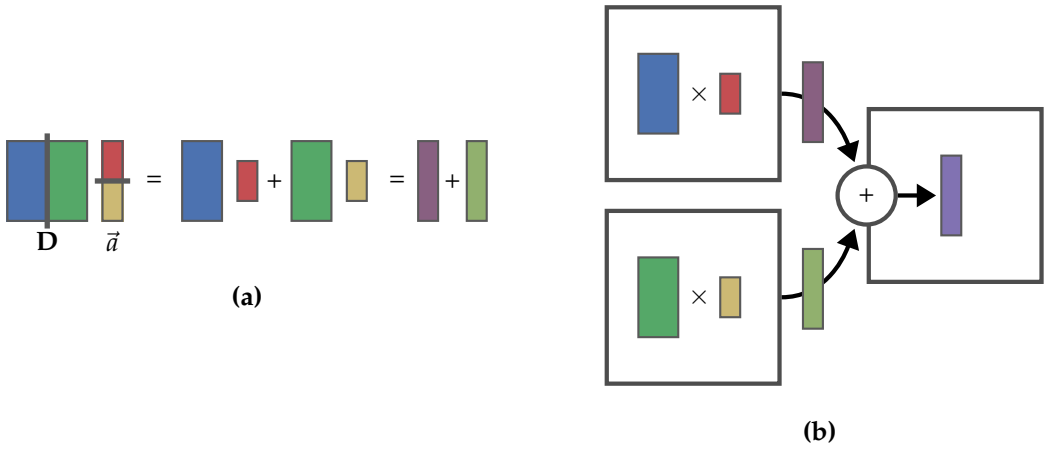


Figure 4.7 – Column-wise division of the decoding operation: (a) shows the series of operations in abstract and (b) shows how the operands are stored and intermediate values transmitted.

If instead of partitioning $\boldsymbol{a}$ we were to store it in shared memory, such that many cores could read and write the entire vector, column-wise partitioning of $\mathbf{D}$ could be replaced by a row-wise partitioning. Figure 4.8 illustrates the revised series of operations and how the data and computation can be mapped onto the SpiNNaker architecture. Row-wise partitioning of $\mathbf{D}$ yields the same memory and compute savings as column-wise partitioning, save for the negligible memory required to store $\boldsymbol{a}$, but does not result in an increase in network traffic.

This technique is possible as long as an ensemble can be partitioned and placed on a single *chip*. To this end, a two-stage partitioning scheme is employed which first breaks an ensemble into *chip*-sized pieces and *then* partitions each sub-ensemble to fit the memory and compute constraints of a single processing *core*. Each chip-sized portion of the ensemble is placed on the same SpiNNaker chip and uses SDRAM to share intermediate values and reduce network traffic. As far as possible the memory and compute load of an ensemble are evenly balanced across the smallest number of chips and cores necessary to simulate it – a typical ensemble in Spaun, consisting of 800 neurons and representing a 16-dimensional space, requires three SpiNNaker cores.

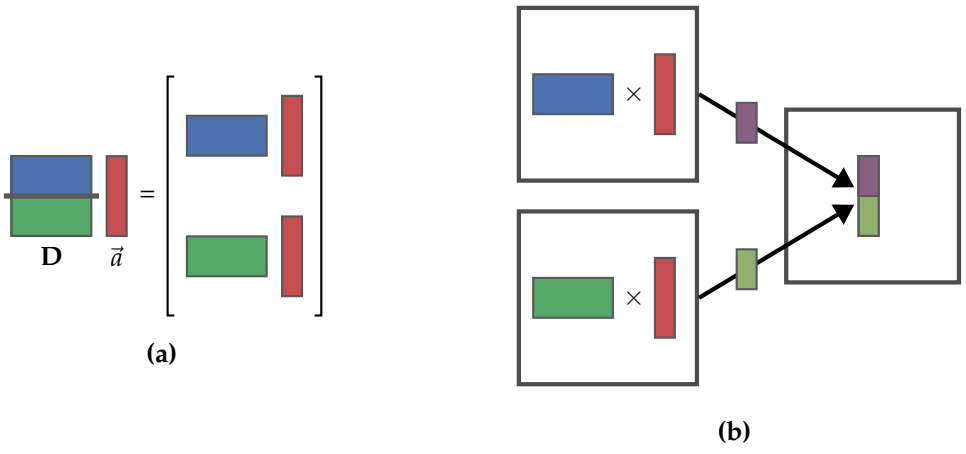


Figure 4.8 – Row-wise division of the decoding operation: (a) shows the series of operations in abstract and (b) shows how the operands are stored and intermediate values transmitted. The mechanism by which $\boldsymbol{a}$ is shared between the transmitting cores is not shown.

Parallel simulation of neurons

As before, neural simulation is split into three stages: *input filtering*, *neuron update* and *output*; however, some tasks are moved between stages and, unlike previously, synchronisation barriers are inserted between stages. Figure 4.9 illustrates this new, parallel, value-based simulation scheme. In the figure, two processing cores work in parallel to simulate a 6-neuron ensemble representing a 2-D space (the encoder matrix is 6×2), the output of which is projected into a 4-D space (the decoder matrix is 4×6). Each processing core is allocated half of the input space to filter, half of the neurons to simulate and half of the output.

Every time step begins with the *input filtering* phase; each core updates its portion of the synaptic filters (in this case the top core filters the first dimension and the lower core the second dimension) and then launches a DMA transfer to copy its portion of the input vector into SDRAM. On completion of the DMA a core waits until all other cores have copied their contribution to the input vector into SDRAM. The *neuron update* stage begins once this synchronisation barrier is passed – each core copies the entire input vector back from SDRAM and simulates its portions of neurons as before. However, instead of immediately decoding firing neurons, a bit-vector is constructed which records the spiking state of each neuron (no spike/spike). Once all neurons have been simulated, the spike vector is transferred into SDRAM to form a complete vector for the entire ensemble and the core again waits for its siblings to reach the synchronisation barrier. Finally, each core copies the complete spike vector from SDRAM and decodes the spikes to produce values of the output vector which are transmitted as multicast packets.

Analysis

In the motivating example we considered the case of two 800-neuron ensembles each representing a 16-D space. Due to memory constraints each ensemble was partitioned across three processing cores with the result that the cores simulating the postsynaptic neurons received 48 packets per time step rather than the 16 which would be necessary had the presynaptic ensemble not been partitioned. We identified that partitioning an ensemble into p parts resulted in a factor p increase in the number of packets required to represent the complete output of that ensemble. We also noted that rapid transmission of the decoded output vector resulted in bursty network traffic.

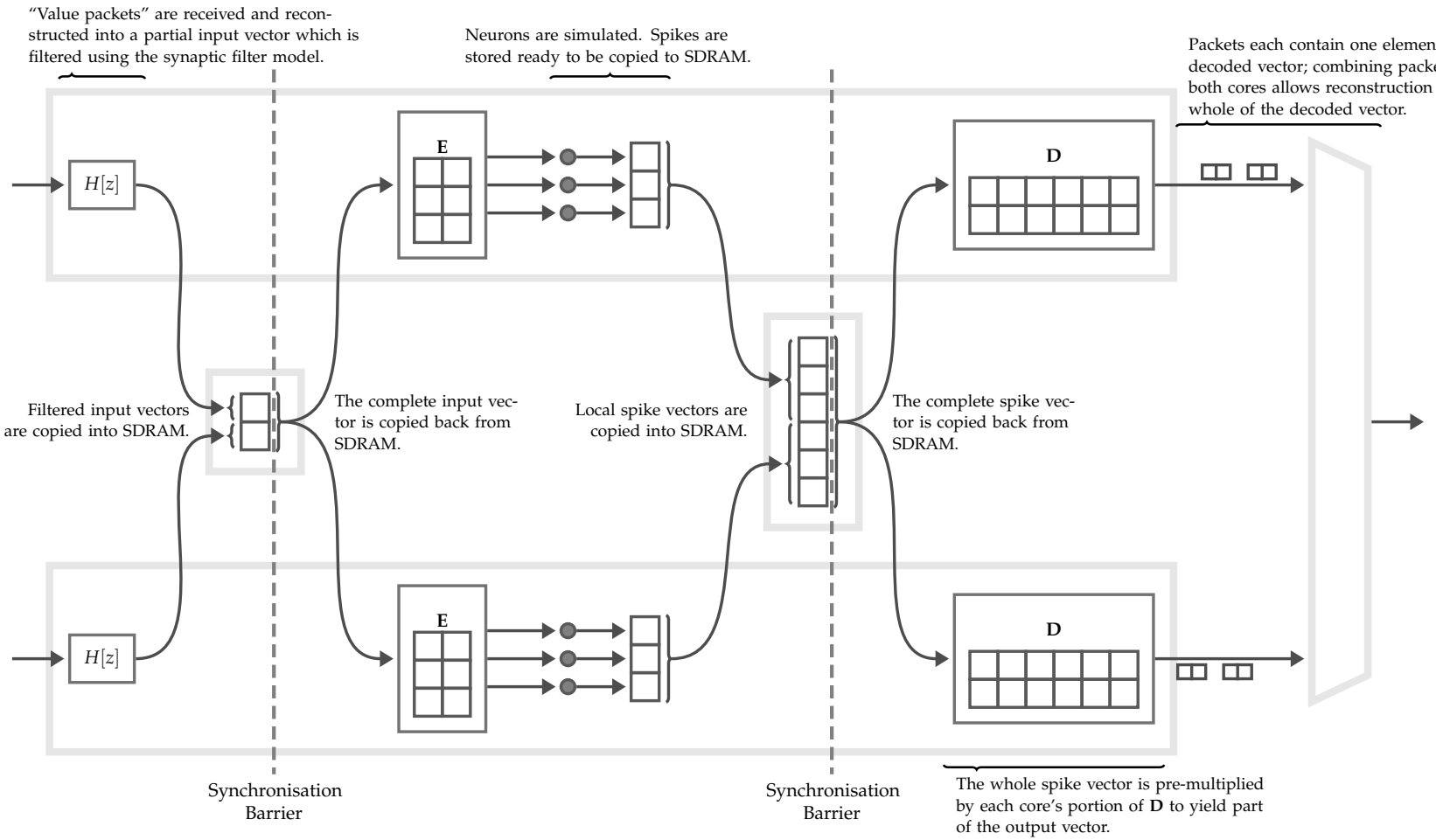


Figure 4.9 – Shared memory simulation of a population of ensembles. The input- and spike-vectors are shared amongst cores on the same chip to reduce the number of multicast packets which must be transmitted to represent the output of the ensemble.

When using memory-sharing, value-based simulation, each of the 800-neuron ensembles would still be partitioned across three processing cores. However, rather than each core transmitting 16 multicast packets per time step, 16 packets are transmitted by the cluster of three cores – with SDRAM being used to share intermediate values (the input and spike vectors) amongst the cores. Additionally, the *output* stage of the simulation step consists of the simultaneous decoding of a spike vector and the transmission of multicast packets, unlike previously where the *output* phase merely transmitted values from a precomputed vector. This new output phase ensures that between transmissions of multicast packets some meaningful computational work is performed, obviating the need to insert a delay to keep multicast packets separated in time.

These gains, however, are offset by the overheads involved in transferring data to and from the shared SDRAM, and in waiting to pass synchronisation barriers. The overheads will be quantified in the subsequent section.

4.4 Performance

Previous sections described the problems which would result from mapping neural networks built using the Neural Engineering Framework to the standard, spike-based, SpiNNaker simulation scheme. To better exploit the SpiNNaker architecture a new, value-based, simulation technique was described and analysed. In this section we shall see that the value-based simulation scheme leads to significant reductions in memory, network and compute load, compared to the spike-based scheme, and is – in some cases – capable of supporting more than 2000 neurons per processing core, double the SpiNNaker architectural target.

Single-core processor utilisation

To evaluate the performance of the ensemble implementation a simple network was simulated while the number of processor cycles spent in each of the simulating stages (*input filtering*, *neuron update* and *output*) was profiled. The network consisted of a single n-neuron ensemble, representing a d-dimensional space, stimulated such that firing rates, and thus computational load, were similar to those that could be expected in an ensem-

ble in Spaun[1]. Input to the ensemble was filtered by a first-order low-pass filter with $\tau = 5\,\text{ms}$. Figure 4.10 shows the distribution of compute costs amongst the three phases of simulation for a range of ensemble scales and representational spaces. The time spent in each phase was computed by multiplying the number of cycles spent in the stage by the time taken per cycle (5 ns for a processor clocked at 200 MHz).

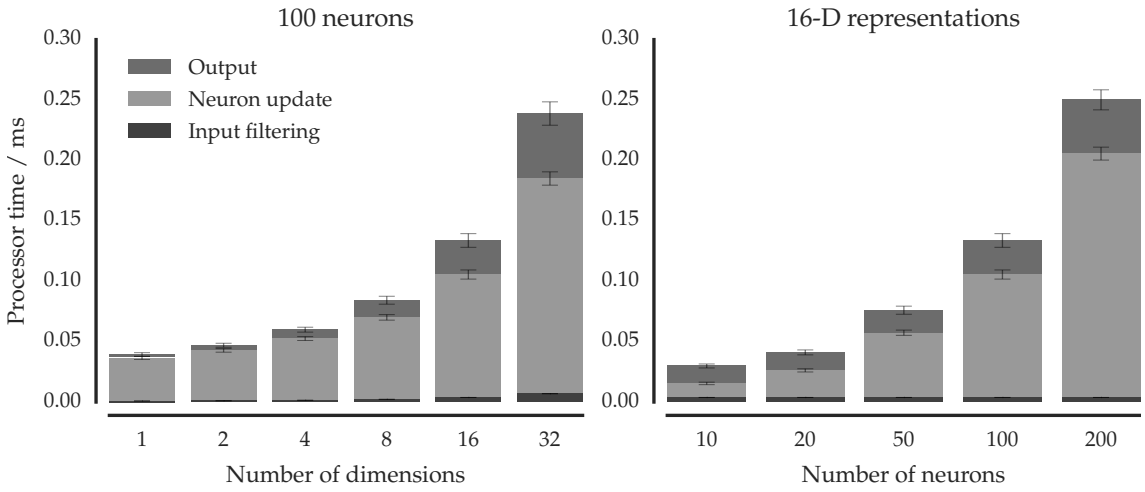


Figure 4.10 – Single-core performance of the ensemble implementation when scaling the number of neurons and the number of represented dimensions. Note that in both cases the horizontal axes are non-linear, see Figure 4.11 for an alternative presentation of these data.

A simple performance model may be fitted to each phase of the simulation. For this model, the compute cost of the *input filtering* stage is a function of the number of dimensions, d, represented by the ensemble. By fitting the parameters of the model to the data we find that the number of cycles used by the *input filtering* stage is:

$$c_{\text{if}} = 39d + 135 \text{ cycles} \tag{4.1}$$

The cost of the *neuron update* phase is a combination of the cost of encoding the input to the neurons (a function of n and d) and the cost of simulating the neurons. As the cost of simulating a neuron depends on whether it is in its refractory period or not the mean across a range of neurons is a useful measure of the work done in simulating an ensemble. Given the input described above, a random walk around the surface of the

[1]Input to the ensemble was a random walk across the surface of the unit d-sphere: this is representative of sampling the firing rate for representing a semantic pointer.

representational sphere, the mean cost of the *neuron update* phase was found to be:

$$c_{\text{nu}} = 9nd + 61n + 174 \text{ cycles} \tag{4.2}$$

The cost of the *output* phase is likewise a function of the number of firing neurons and the number of dimensions in the decoding. Using the same techniques as before, the mean cost of this phase was found to be:

$$c_{\text{out}} = 2nd + 143d + 173 \text{ cycles} \tag{4.3}$$

Hence, the total compute cost for this configuration of the ensemble can be expressed as:

$$c_{\text{total}} = 11nd + 61n + 182d + 482 \text{ cycles} \tag{4.4}$$

When simulation steps are 1 ms long, and the processor is clocked at 200 MHz, a total of 2×10^5 cycles are available per time step. Consequently, with $d = 1$, up to 2700 neurons may be allocated to each processing core – more than twice as many as the architectural target of 1000 neurons per core (Furber and Temple 2007). Figure 4.11 illustrates how the models derived above fit the recorded profiling data.

By substituting $d = \frac{n}{50}$ to reflect the common-case in Nengo simulations we can further simplify the total cost to:

$$c_{\text{total}} = 0.22n^2 + 64.64n + 482 \text{ cycles} \tag{4.5}$$

Using the cost estimates for synaptic processing and neuron update of Sharp and Furber (2013), we expect the spike-based scheme to have the cost:

$$c_{\text{spikes}} = 3n^2 + 128n \text{ cycles} \tag{4.6}$$

These models show that value-based simulation can, in this configuration, allow for simulation of up to 815 neurons per core, more than three times as many as the 237 neurons per core possible with the spike-based scheme in the same configuration.

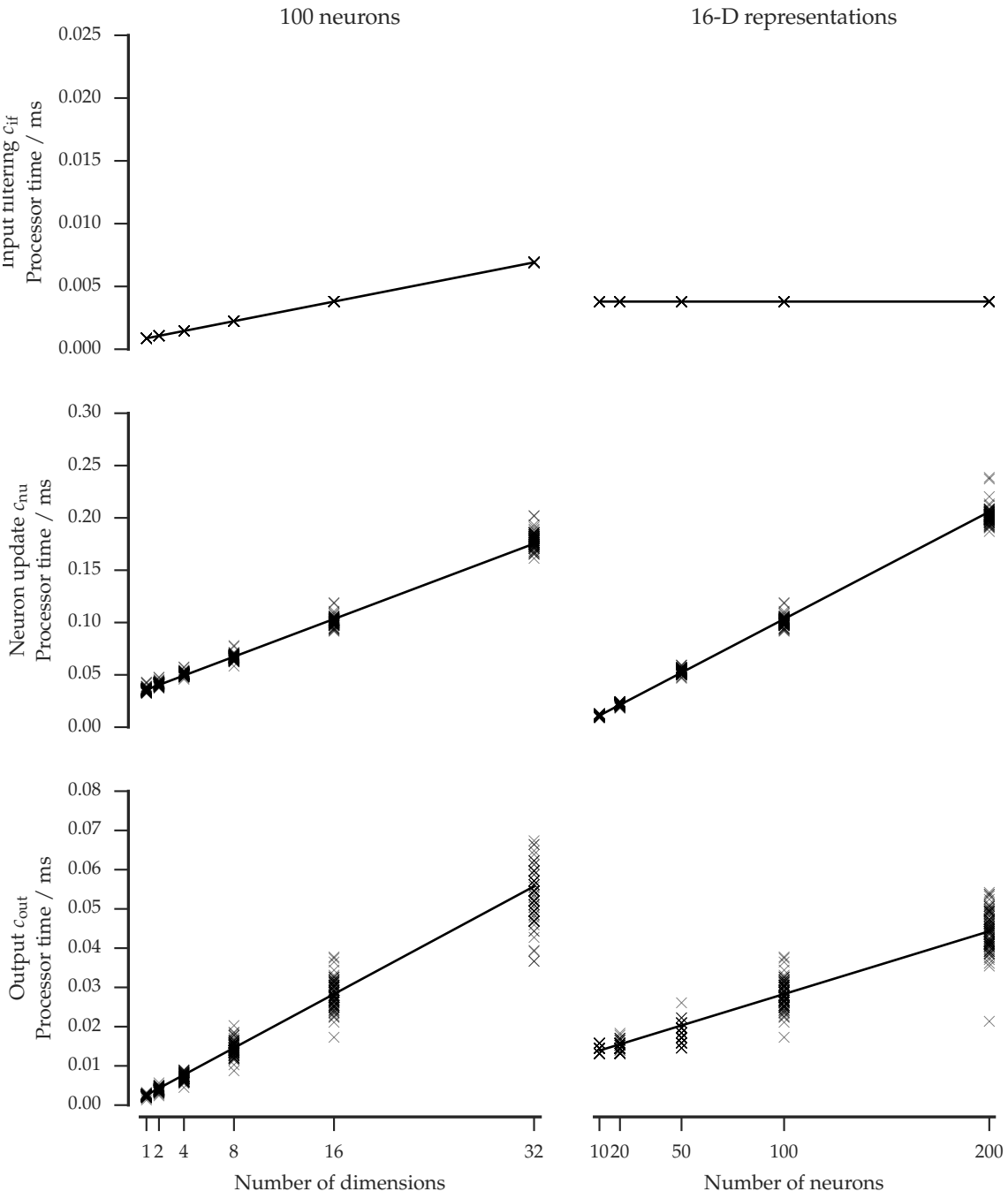


Figure 4.11 – Modelled and recorded ensemble performance. Each plot shows a sample of the measured ensemble performance plotted against the models that were fitted above. For each data point (for example, 16-D at 200 neurons) 100 of the 10 000 recorded samples are plotted. It is worth noting that as the time spent in the *neuron update* and *output* phases is a function of the number of neurons which spiked in the preceding time step the variance of the cost of these phases increases with the number of neurons.

Multiple-core processor utilisation

The above results related only to a single processor and so did not include the overheads of writing and reading shared data to SDRAM and synchronisation incurred by the shared-memory simulation scheme. To find these overheads a similar experiment to that in the previous section was performed, but with the ensembles scaled to require multiple cores. It was found that the transition from the *neuron update* phase of processing to the *output phase* – a transition which includes writing spike values to SDRAM, synchronising and reading from SDRAM – was less than 0.1 ms when using 8 processors in parallel to simulate 200, 400 or 800 neurons. The overheads resulting from shared memory neural simulation are therefore small relative to the time required to simulate the neurons.

Packet processing cost

The computational work of a processor is a combination of the various simulation phases *and* the tasks performed to process received packets – see Section 4.2 (p. 77). To ensure that SpiNNaker cores are not overloaded the partitioning of ensembles over processors must be informed both by the performance models of Section 4.4 and a model of the work involved in receiving packets. To build this model these latter costs were profiled by simulating a network and measuring the time taken by one core to handle the packets it received. Additional entries were inserted into the filter routing table (see Figure 4.6, page 79) to allow time taken to compare a packet against an entry and that taken to include the contribution of a matching packet in the input to a filter to be measured. Figure 4.12 illustrates a modified filter routing table and shows the results of the experiment.

These measurements were used to create a model of the processor cycles taken to process a packet:

$$c_{\text{mc}} = 12(m + n) + 16m + 49 \text{ cycles} \tag{4.7}$$

Where m is the number of entries which match against the packet and n is the number of entries which do not. In Chapter 5 this expression is used to determine how many cores are required to simulate large neural models.

Although this expression could have been derived by inspecting the assembly code use of profiling data captures more information about on-chip performance, for exam-

ple the cost of taking different branches during execution. For these measurements the largest observed standard deviation occurred for $m = 1$, $n = 0$ and was less than 2 cycles, other data points had standard deviations of the order of 10^{-8} cycles, suggesting very occasional differences in performance.

Network loading

Section 4.3 used the example of the network traffic resulting from a 16-D decoding of an 800-neuron ensemble to motivate the use of SDRAM to share intermediate values amongst processors. In the example it was noted that the use of shared memory would reduce the number of packets transmitted per time step from 48 to 16, despite the same number of cores being required to simulate the ensemble. Figure 4.13 shows the results of an experiment to demonstrate this effect. Measurements of the router packet counters were made at 5 µs intervals for a period of 10 ms. For the ten time steps shown, 48 packets were transmitted in each burst of activity when not using shared memory compared to 16 when intermediate results were shared.

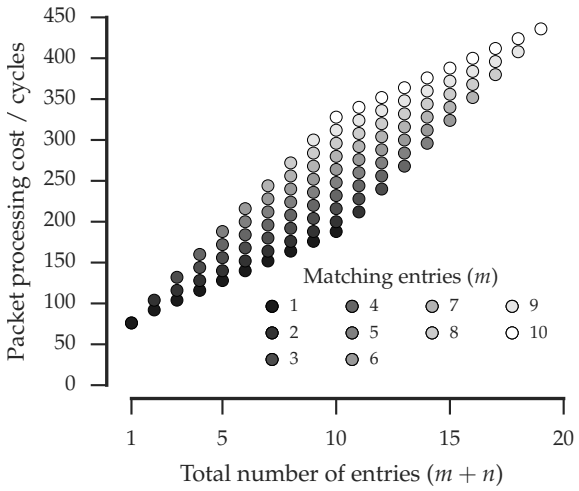
An additional motivation for the use of shared-memory was to decrease the burstiness of the traffic. The mean burst duration when not using shared memory was found to be approximately 30 µs, resulting in a burst bandwidth of 115 bit μs^{-1}. When shared-memory parallelism was used the mean burst duration increased to around 55 µs, lowering the burst bandwidth to 21 bit μs^{-1}. Use of shared-memory parallelism thus leads to around a factor five reduction in the network load associated with this simple model.

4.5 Correctness

To validate the correctness of the SpiNNaker the implementation of the Neural Engineering Framework a series of experiments is presented below. These begin by testing the implementation of the leaky-integrate and fire neuron model at a range of frequencies before assessing the correctness of the implementation with respect to the three principles of the NEF: representation, transformation and dynamics. The intent of these experiments is not to guarantee that SpiNNaker behaviour will always match that of Nengo but to demonstrate that the simulation techniques described in this chapter produce reasonable neural behaviour.

Key	Mask	...	Filter
`0x00000000`	`0xffffff00`		0
`0x00000000`	`0x00000000`		0
`0x00000000`	`0x00000000`		0
`0xffffffff`	`0x00000000`		–

(a)



(b)

Figure 4.12 – The packet processing cost was measured by inserting additional entries into the filter routing table (see Figure 4.6), (a) shows a table in which the first entry matches a specific set of packets, the next two entries are "match-all" entries and the final entry is a "match-nothing" entry ($m = 3, n = 1$). The processor cycles spent in the packet handler for various m and n is shown in (b), each point is the mean of 1000 samples, c.f. equation (4.7). The ordering of the entries does not affect the time required to process incoming packets.

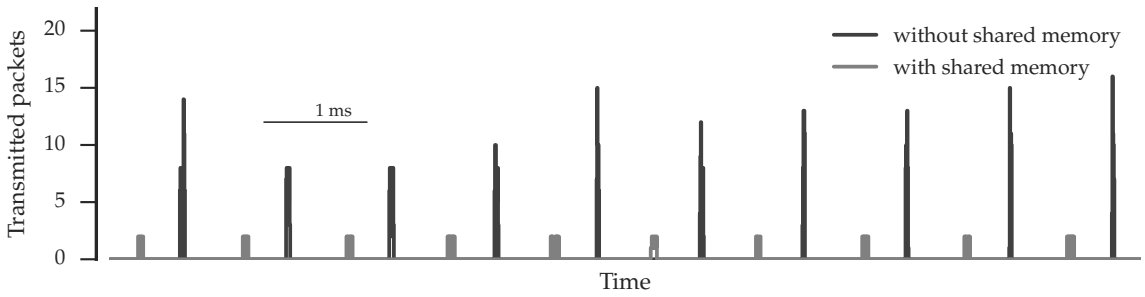


Figure 4.13 – Packets transmitted for a 16-D decoding of an 800-neuron ensemble. Router packet counters were sampled at 5 µs intervals during simulation of ensembles using both the original value-based technique (Section 4.2) and the shared memory technique (Section 4.3). When using the shared memory technique fewer packets were transmitted overall and these were spread over more time – leading to a reduction in network load.

Neural tuning curves

The tuning curve of a neuron indicates the frequency of the firing response of a neuron to a given input. Figure 4.14 shows both the 'ideal' tuning curve (as derived analytically by Eliasmith and Anderson 2004, p.37) and the SpiNNaker equivalent for a LIF neuron with $\tau_{\text{ref}} = 2\,\text{ms}$ and $\tau_{\text{rc}} = 20\,\text{ms}$. The lower part of the same plot shows the percentage error between the Nengo reference implementation, the SpiNNaker implementation and the ideal. Both the reference implementation and SpiNNaker tend to fire slower than the ideal model, with the SpiNNaker implementation firing slightly slower than the reference. The effect of this will be to slightly reduce the representational accuracy of ensembles built using the neurons; this will be quantified by subsequent experiments.

Representation

To test the processes of encoding and decoding values, a single ensemble of 100 LIF neurons was configured to represent a scalar value. Over the course of 50 tests an input value between ± 1 was applied to the ensemble for the duration of 10 s and the decoded output of the ensemble was filtered by a first-order low-pass filter with a 5 ms time constant and recorded. The output of one such test, for both the reference implementation and SpiNNaker, is shown in Figure 4.15(a). For each test the Root Mean Square Error (RMSE) between both the reference and SpiNNaker instantiations of the model and the ideal output was computed and the results are shown in Figure 4.15(b). For these samples, the SpiNNaker instantiation suffers from a similar level of error to the reference implementation, indicating that the encoding and decoding schemes allow SpiNNaker to simulate simple networks accurately.

Transformation

By modifying the decoders of an ensemble we can compute functions of the value represented by an ensemble. This transformation principle of the NEF can be demonstrated on SpiNNaker with a slight change to the neural model, in this case such that the decoders act to compute the square of the represented value. Figure 4.16(a) shows sample input and output to such a network. As before, a number of experiments were run to determine the error between the output of the simulated ensemble (running on both SpiNNaker and the reference simulator) and the ideal output. Figure 4.16(b) shows the results of these

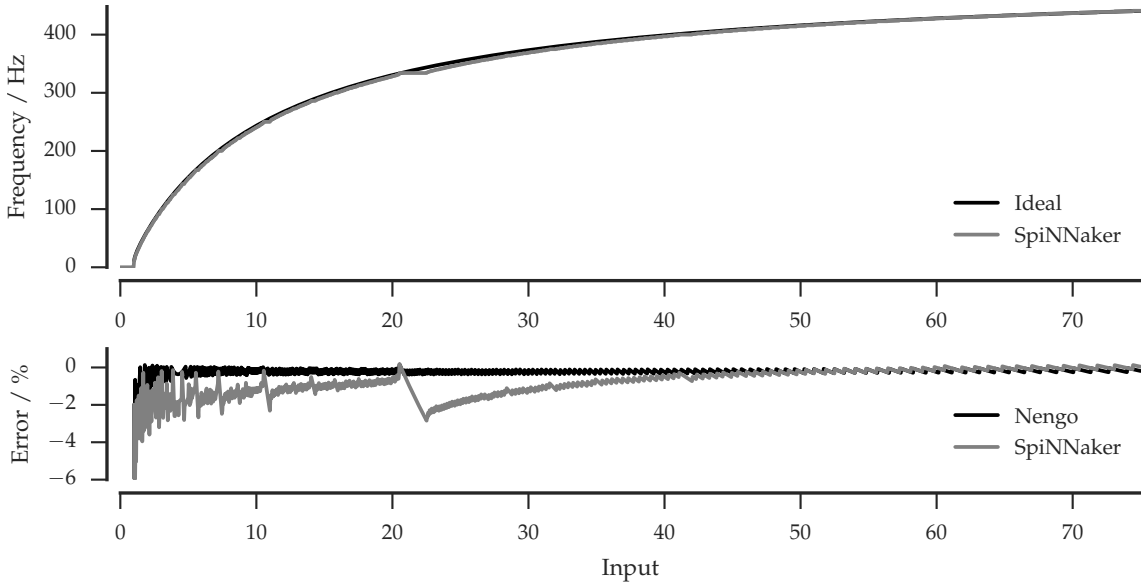


Figure 4.14 – Tuning curve of a LIF neuron as implemented on SpiNNaker. The relative error between the ideal curve and the reference and SpiNNaker implementations is shown in the lower plot.

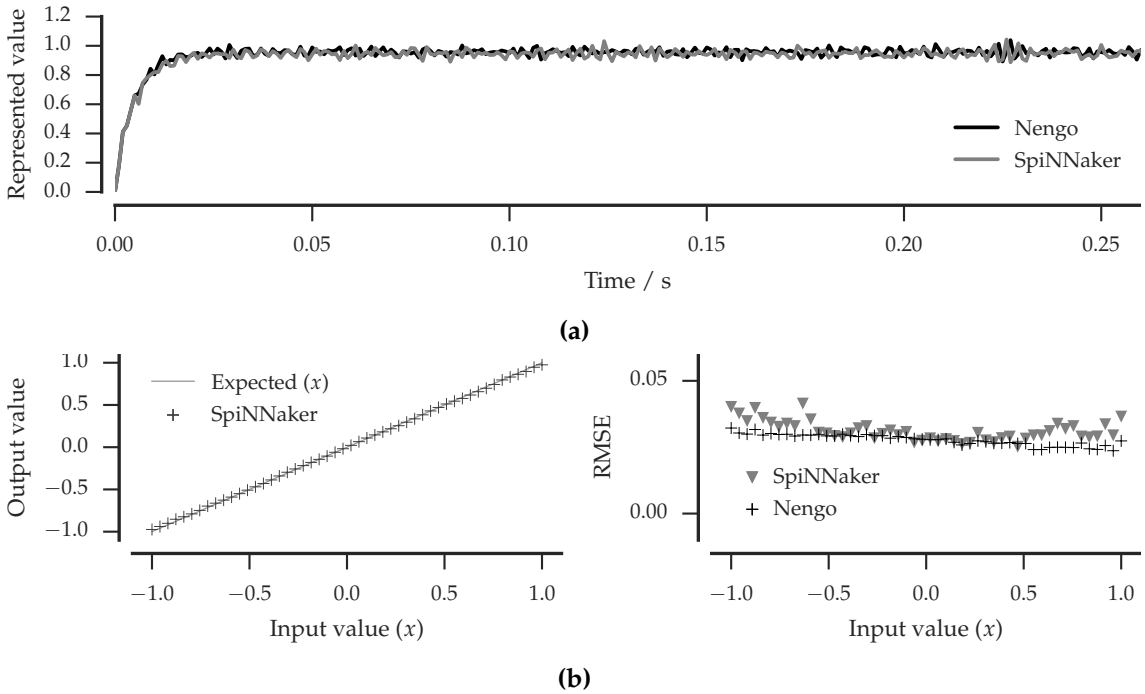


Figure 4.15 – Representing values with neurons on SpiNNaker. (a) shows the output of a population of 100 neurons representing a constant value. This ensemble was simulated with both Nengo and SpiNNaker. (b) shows how the error between the expected value and that decoded from the neurons varies for both Nengo and SpiNNaker. The left hand plot shows the outputs produced by SpiNNaker as compared to the expected, the right hand plot shows the Root Mean Square Error (RMSE) for both platforms.

experiments. Again, SpiNNaker is shown to perform as well as the reference implementation of the NEF despite the need to use fixed point representations.

Dynamics

Finally, the Neural Engineering Framework may be used to construct dynamic systems (Section 2.2). Figure 4.17 shows the behaviour of a neurally-implemented integrator when simulated both on SpiNNaker and with the reference simulator. While the output of the SpiNNaker instantiation is similar to the output of the reference simulator it should be noted that the SpiNNaker implementation of the integrator tends to "leak" slightly (note the decreasing value from 1 s to 2 s). This is probably due to insufficient precision being available to represent the coefficients of the synaptic filter on the recurrent connection.

4.6 Summary

The dense synaptic weight matrices and high firing rates characteristic of neural networks built using the Neural Engineering Framework (NEF) lead to inefficient use of the SpiNNaker architecture when using the standard algorithms for simulating neural nets. In particular storing the synaptic matrices of these networks requires large amounts of memory and the high firing rates and dense neural connectivity exceed the computational resources available to a SpiNNaker core running in biological real time. To overcome these challenges to the use of the Neural Engineering Framework on SpiNNaker a value-based simulation scheme was developed, extending the work of Bekolay et al. (2014). This value-based scheme has been shown to result in around a 90 % reduction in the memory usage associated with a range of simulation scales, and makes considerably better use of the computation and communication resources of the SpiNNaker chip – to the extent that in some cases twice as many neurons may be simulated on a core than was the architectural target.

This chapter resolved some of the problems associated with SpiNNaker simulation of neural networks built with the NEF, in particular the computation and memory requirements. Challenges still remain, however, with the *communication* required to simulate large neural models and this will be addressed in the next chapter.

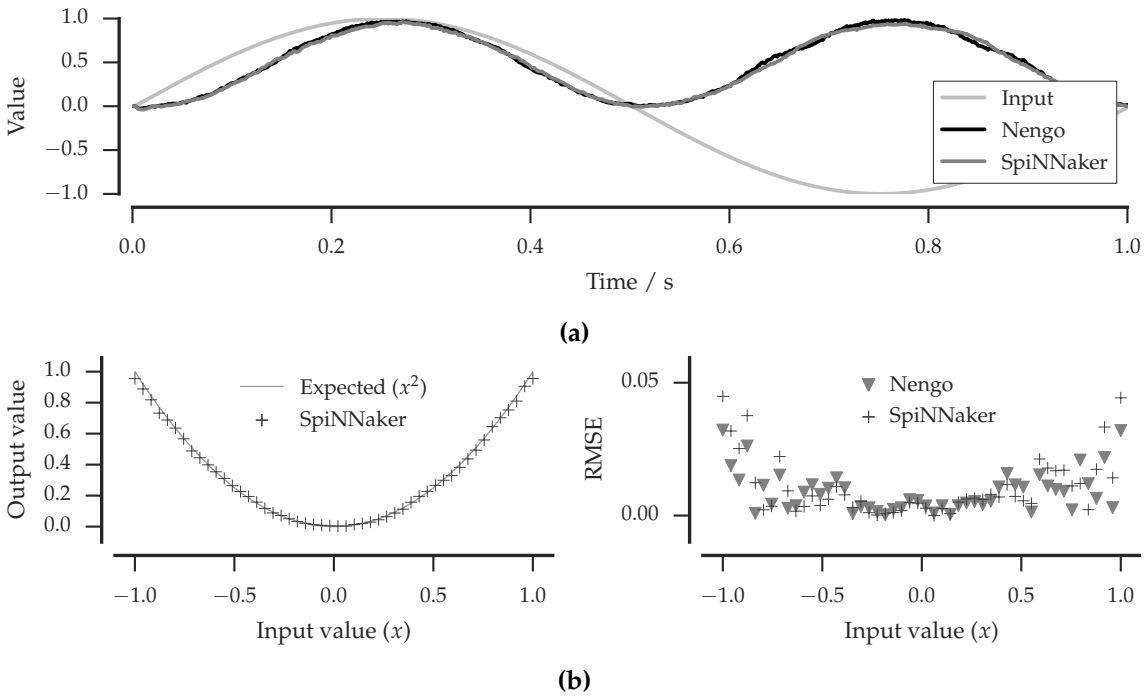


Figure 4.16 – Transforming values with neurons on SpiNNaker. (a) shows the input to and output of a population of 100 neurons representing a changing value. The decoders of the ensemble have been selected such that the output represents an estimation of the square of the input. (b) shows how the error between the expected and decoded values varies for both Nengo and SpiNNaker when computing the square of an input representation. The left hand plot shows the outputs produced by SpiNNaker as compared to the expected, the right hand plot shows the Root Mean Square Error (RMSE) for both platforms.

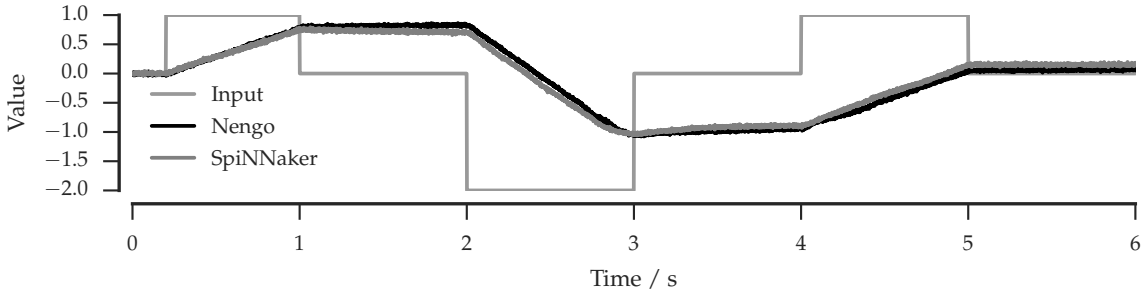


Figure 4.17 – Sample output of a neural integrator implemented with the NEF

Notes

The value-based scheme presented in this chapter was extended by Knight, Voelker, et al. (2016) to support both supervised and unsupervised learning rules. Additionally, the SpiNNaker simulator developed in this chapter was used by Stewart, Kleinhans, et al. (2016) to investigate learning in a neuromorphic robot.

Chapter 5

The Semantic Pointer Architecture and SpiNNaker

The previous chapter investigated the problem of simulating neural networks built using the Neural Engineering Framework on the SpiNNaker architecture. While the solution to the problems identified significantly improved the use of the platform made by this class of neural nets, the challenge for SpiNNaker is to facilitate accurate simulation of neural models of significant scale in biological real time. The previous chapter demonstrated correct simulation of small neural networks; the few benchmarks used also ran successfully on an earlier implementation of the NEF for SpiNNaker (Galluppi et al. 2012). This chapter investigates the techniques necessary to allow simulation of large neural models, approaching the scale of Spaun.

5.1 Representing high-dimensional values

In Section 2.3 we saw that we could represent token-like symbols as high dimensional vectors and that addition and circular convolution could be used to construct new vectors to represent compound combinations of symbols. The Neural Engineering Framework (Section 2.2) describes a way in which populations of neurons and the connections between them can represent and manipulate these vectors. To decrease the likelihood of confusing two vectors they are drawn from a high dimensional space. Symbols in Spaun, for example, are represented by vectors in a 512-dimensional space and Crawford, Gingerich, and Eliasmith (2013) argue that this is sufficient to represent a human-

scale lexicon. For an ensemble to represent a high dimensional space accurately it must contain a large number of neurons – between $50d$ and $70d$ neurons for an d-dimensional space (Eliasmith 2013). However, as the number of neurons in an ensemble increases the computational cost of finding appropriate decoders also increases. To avoid costly decoder selection, models containing ensembles which need to represent high dimensional spaces often break the space into smaller chunks which can be represented by smaller ensembles (Gosmann and Eliasmith 2016).

For example, Figure 5.1(a) shows a simple network containing two ensembles each representing a two-dimensional value. The first ensemble represents $\boldsymbol{x}$ and the second $\boldsymbol{y}$ and they are connected such that $\boldsymbol{y} = \boldsymbol{x}$. Since x and y are both 2-dimensional vectors the identity matrix, $\mathbf{I}$, is applied by the connection between them. In Figure 5.1(b) the ensembles are each replaced by two smaller ensembles. These smaller ensembles each represent one dimension of the values represented by the larger ensembles, i.e., $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$. Since $\boldsymbol{y} = \mathbf{I}\boldsymbol{x}$ we have $\boldsymbol{y}_1 = 1 \times \boldsymbol{x}_1 + 0 \times \boldsymbol{x}_2$ and $\boldsymbol{y}_2 = 0 \times \boldsymbol{x}_1 + 1 \times \boldsymbol{x}_2$, and no connections are required between the ensembles representing $\boldsymbol{x}_1$ and $\boldsymbol{y}_2$ or $\boldsymbol{x}_2$ and $\boldsymbol{y}_1$. In the same way, the 512-dimensional spaces in Spaun are not represented by single ensembles of 25 600 neurons but are instead represented by groups of thirty-two 800-neuron ensembles each representing a 16-dimensional subspace[1].

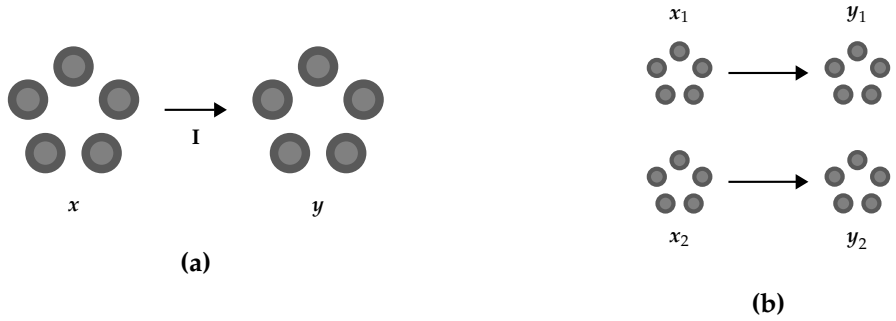


Figure 5.1 – Splitting large ensembles to reduce decoder compute time. In (a) two ensembles, each representing a 2-dimensional space, are shown connected together. These ensembles may each be replaced by two smaller ensembles, each representing one of the two dimensions represented by the original ensembles, as shown in (b).

Not only does breaking larger ensembles apart significantly reduce the time required to choose decoders but this pragmatism in model construction also benefits SpiNNaker

[1] This is the default as selected by Nengo (https://github.com/nengo/nengo/blob/master/nengo/spa/state.py – accessed January 9, 2017)

simulation of the network. To represent a 512-dimensional space using the heuristic given above would require a 25 600-neuron ensemble. The encoders and decoders of such an ensemble would consume $2 \times 25\,600 \times 512 \times 4\,\mathrm{B} = 100\,\mathrm{MiB}$. Since, in the simulation schemes described above, the encoders and decoders are stored in the DTCMs of processing cores an ensemble of this size would need to be partitioned over $\lceil \frac{100\,\mathrm{MiB}}{64\,\mathrm{KiB}} \rceil = 1600$ cores, more than 90 SpiNNaker *chips*. In Spaun, however, a 512-dimensional space is instead represented by thirty-two 800-neuron ensembles each representing a 16-dimensional space. Only 100 KiB is required to store the combined encoders and decoders for one of these smaller ensembles. While one of these ensembles could be simulated using two SpiNNaker cores, in practice they are partitioned over three cores each to ensure adequate space remains for other data structures. Thus, while simulating the 'large' ensemble would require the use of more than 90 *chips*, thirty-two of the 'smaller' ensembles need only 96 *cores* (less than 6 chips). In this case using smaller ensembles achieves a 94 % saving in the number of cores required.

However, the example shown in Figure 5.1 is not the general case. In Figure 5.2(a) the connection between the two larger ensembles has been modified to apply a linear transform such that $\boldsymbol{y} = \mathbf{L}\boldsymbol{x}$ (see Section 2.2). The transform, $\mathbf{L}$, might, for example, represent a rotation around the origin by the angle θ (an example we saw earlier):

$$\mathbf{L} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

When the ensembles representing $\boldsymbol{x}$ and $\boldsymbol{y}$ are split more connections are required between the resulting, smaller, ensembles. Since $\boldsymbol{y}_1 = \boldsymbol{x}_1 \cos\theta - \boldsymbol{x}_2 \sin\theta$ the ensemble representing $\boldsymbol{y}_1$ must receive input from *both* the ensembles representing $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. Likewise, since $\boldsymbol{y}_2 = \boldsymbol{x}_1 \sin\theta + \boldsymbol{x}_2 \cos\theta$ it too must receive input from *both* the presynaptic ensembles. The resultant, denser, connectivity is shown in Figure 5.2(b).

Since the cores simulating $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ both receive input from $\boldsymbol{x_1}$, the latter will transmit two packets per time step representing the values $\boldsymbol{x}_1 \cos\theta$ and $\boldsymbol{x}_1 \sin\theta$ to cores $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ respectively. Likewise, the processing core simulating $\boldsymbol{x}_2$ will transmit two packets per time step representing $-\boldsymbol{x}_2 \sin\theta$ and $\boldsymbol{x}_2 \cos\theta$. Consequently, the processing cores simulating ensembles $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ each receive two packets per time step. Since the packets transmitted by the cores simulating $\boldsymbol{x}$ represent different values ($\boldsymbol{x}_1 \cos\theta$, $\boldsymbol{x}_1 \sin\theta$ etc)

they *cannot be multicast* by the SpiNNaker network. Hence the total amount of traffic is increased; as the ensembles are split further more traffic results.

Consequently, while the splitting of large ensembles is beneficial in reducing the number of processing cores needed to simulate a given network it implies that, in the general case, the SpiNNaker network will be more heavily used. The likelihood of the network becoming congested and dropping packets, reducing the accuracy of the simulation, is directly linked to the traffic. In addition, since in the general case each smaller ensemble receives input from a greater number of other ensembles the processing core simulating such an ensemble will receive a greater number of multicast packets, increasing its computational load and the likelihood that a real time deadline will be missed. Since any dropped packets or missed deadlines will reduce the accuracy of a simulation, eliminating such events is crucial if SpiNNaker is to be used for the simulation of large scale neural models.

Large or small ensembles?

Although, as seen before, fewer processing cores are required to simulate networks consisting of smaller ensembles than those consisting of larger ones this seems to come at the cost of greater network connectivity. Hence it is worth assessing whether the technique of splitting larger ensembles into groups of smaller ensembles should be abandoned.

To be simulated on SpiNNaker, a 25 600-neuron ensemble, representing a 512-dimensional space, would be partitioned across more than ninety chips to satisfy memory con-

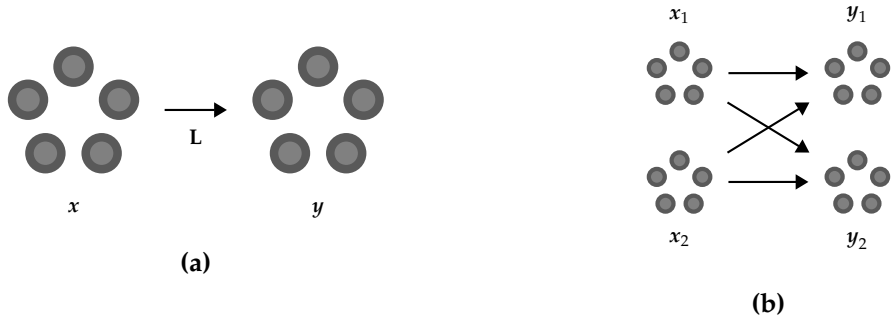


Figure 5.2 – Replacing larger ensembles with groups of smaller ones can increase the connectivity between groups of ensembles. In (a) two ensembles are connected such that the second represents a transformation of the value from the first. Since this transformation spreads information between subspaces greater network connectivity is required, as shown in (b). This is the general case of Figure 5.1.

straints. Under the shared memory simulation scheme presented in Section 4.3 each *chip* would transmit 512 packets every simulation time step. Consequently, any core receiving input from this large ensemble would receive more than $90 \times 512 = 46\,000$ packets per time step. Given that receiving and processing a packet costs at least 77 cycles (see Eq. 4.7 (p. 89), for $m = 1$ and $n = 0$) receipt of 46 000 packets would require more than 3.5 million cycles. Since this is 17 times more cycles than are available to a processing core when simulating 1 ms time steps in real time and clocked at 200 MHz, simulation would need to proceed at least 17 times slower than biology.

Figure 5.2 shows what happens to network connectivity when ensembles are split. In the case that $\boldsymbol{x}$ and $\boldsymbol{y}$ represented 512-dimensional spaces they would be split, as described before, into groups of thirty-two 800-neuron ensembles representing 16-dimensional subspaces. Since each ensemble in $\boldsymbol{y}$ would represent 16-dimensions it would receive 16 packets every time step from each of the 32 ensembles in $\boldsymbol{x}$. Again assuming that 77 cycles are required to receive and process each packet, a total of $77 \times 16 \times 32 = 39\,424$ cycles would be consumed. Since this is less than 20 % of the 200 000 cycles available to a core simulating a 1 ms time step the simulation can proceed in biological time.

Since the technique of breaking larger ensembles into smaller ones can be seen not only to reduce the number of processing cores required but also to allow for simulation in biological real time it should not be abandoned. However, the network traffic and, particularly, the computational cost associated with receiving packets are high and will become worse when larger and more densely connected networks are considered. This is problematic since the traffic and compute requirements are likely to lead to missed deadlines and network congestion, resulting in poor simulation accuracy.

Techniques for improving reliability

There is a number of ways in which the problems arising from the need to transmit and receive high dimensional values might be addressed. The simplest would be to increase the time allowed to perform each step of the simulation while keeping the same time step. This would give each core more time to process the packets it receives, to simulate the neurons it has been allocated and to spread out the packets it transmits, simultaneously reducing network congestion and the likelihood of missing a deadline at no cost to the fidelity of the simulation. However, this would mean that the simulation was running

slower than biology. Since this would rule out a large range SpiNNaker applications it shall be disregarded for now but it does remain an option for applications where slower than real time simulation is acceptable.

Alternatively, coarser simulation time steps could be used while continuing to simulate at biological real time. This would increase the compute time allowed for each time step, resulting in the same benefits as before, but would come at the cost of accuracy. Moreover, reducing the granularity of the time step would limit the range of neural models which could be simulated.

Finally, if dense connectivity of the form described can be identified while preparing the model for simulation then an interposer could be added to the SpiNNaker instantiation of the model. Figure 5.3 illustrates this technique. This additional component, which has no direct biological counterpart, exists solely in the SpiNNaker instantiation and acts to reduce the fan-out and fan-in of the processing cores with which it communicates.

5.2 Interposer design

Dense network connectivity between groups of ensembles representing high dimensional spaces occurs when a transform is applied to the value decoded from one of the groups of ensembles. We have some choice about *where* this transform is applied; in Figure 5.2(b) (p. 100) the transform was applied by the processing cores simulating the ensembles representing $\boldsymbol{x}$ (specifically, during preparation of the network the decoders of the ensembles were premultiplied by the transform, as described in Section 2.2). Since the application of a transform in this way increases the network connectivity we will instead introduce an *interposer* to apply the transform. This new component must perform an efficient matrix-vector multiplication. There are several ways this might be implemented (e.g., Kung and Leiserson 1980) but we will investigate performing decompositions of the matrix in a manner similar to Strassen's algorithm (Cormen 2009).

Figure 5.3(a) shows a neural network made up of two groups of four ensembles. These groups of ensembles work together to represent two 8-dimensional values, $\boldsymbol{x}$ and $\boldsymbol{y}$, and the groups are connected together such that $\boldsymbol{y} = \mathbf{L}\boldsymbol{x}$ where $\mathbf{L}$ is an 8×8 matrix. The ensembles in a group each represent a 2-dimensional subspace of the space represented by the group. For example, the ensemble marked $\boldsymbol{x}_{3,4}$ represents the third and fourth dimensions of the vector $\boldsymbol{x}$. With no interposer, assuming each ensemble can be

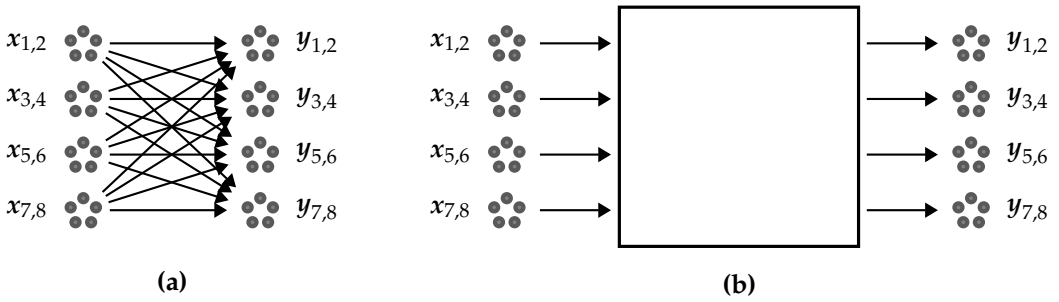


Figure 5.3 – Use of an interposer to decrease fan-out, fan-in and overall traffic. A neural network consisting of two groups of ensembles is shown in (a). For simulation purposes, the dense connectivity between the two groups of ensembles is replaced by a non-neural interposer in (b) which decreases the fan-out of the ensembles to the left and the fan-in of the ensembles to the right.

simulated by a single processing core, two (non-multicast) packets per simulation time step are used to transmit the values represented by each connection. For example, the connection between the ensembles marked $\boldsymbol{x}_{1,2}$ and $\boldsymbol{y}_{1,2}$ carries the values $\mathbf{L}_{11}\boldsymbol{x}_1 + \mathbf{L}_{12}\boldsymbol{x}_2$ and $\mathbf{L}_{21}\boldsymbol{x}_1 + \mathbf{L}_{22}\boldsymbol{x}_2$. Consequently, each processing core simulating an ensemble marked $\boldsymbol{x}$ transmits eight different packets per step of the simulation and eight are received by each ensemble marked $\boldsymbol{y}$.

When an interposer is added to the SpiNNaker network the number of packets transmitted and received by each processing core is reduced. Figure 5.3(b) illustrates this case, where the processing cores simulating ensembles marked $\boldsymbol{x}$ each transmit two packets per simulation time step and processing cores simulating the ensembles marked $\boldsymbol{y}$ each receive two packets. For example, the core simulating the ensemble $\boldsymbol{x}_{1,2}$ would transmit packets containing the values $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. These packets would be combined in the interposer to form a vector, $\boldsymbol{x}$, which would then be premultiplied by the matrix $\mathbf{L}$ and a further series of packets sent to transmit the resulting, transformed, vector. This, of course, simply moves the problem of performing the matrix-vector multiply and transmitting the result into the interposer 'box' but this can provide significant benefits, as seen below.

For a small example, such as this, the interposer could be implemented with a single processing core, as illustrated in Figure 5.4. However, this does not scale well and – in a similar way to the ensembles – it is expedient to partition the matrix-vector multiply.

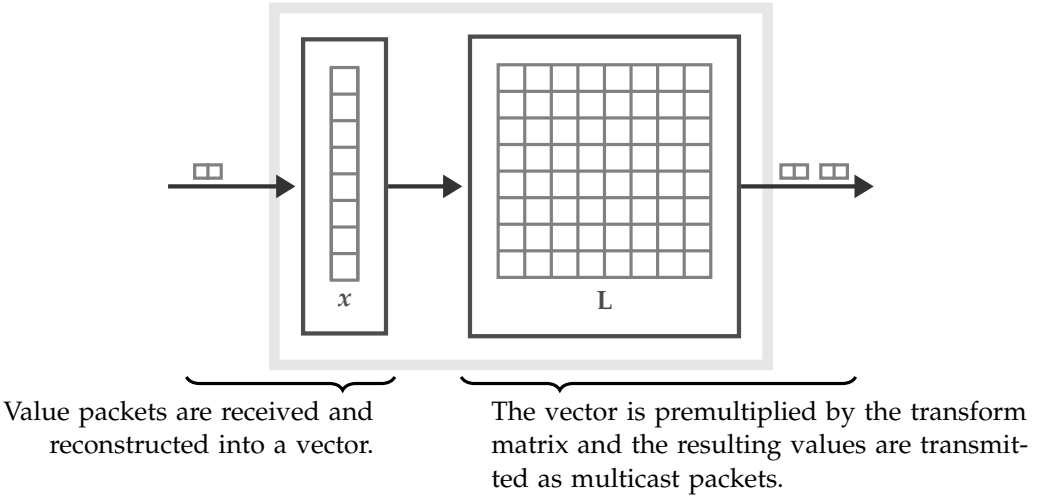


Figure 5.4 – Implementation of an interposer core. Received multicast packets are formed into a vector which is premultiplied by the matrix **L** to form an output vector which is transmitted with further packets.

Row partitioning

Figure 5.5 illustrates the division of the 8×8 matrix **L** into four partitions. Each of these 2×8 partitions is allocated to a single processing core (implemented as shown in Figure 5.4) and the network connectivity is modified to ensure that the necessary values are delivered to the correct processing cores.

Since the processing cores 'A', 'B', 'C' and 'D' receive the same values (representing $\boldsymbol{x}$) the packets representing these values may be *multicast*, making effective use of the SpiNNaker network. Consequently, where each processing core simulating an ensemble labelled $\boldsymbol{x}$ needed to transmit eight packets per simulation step when an interposer

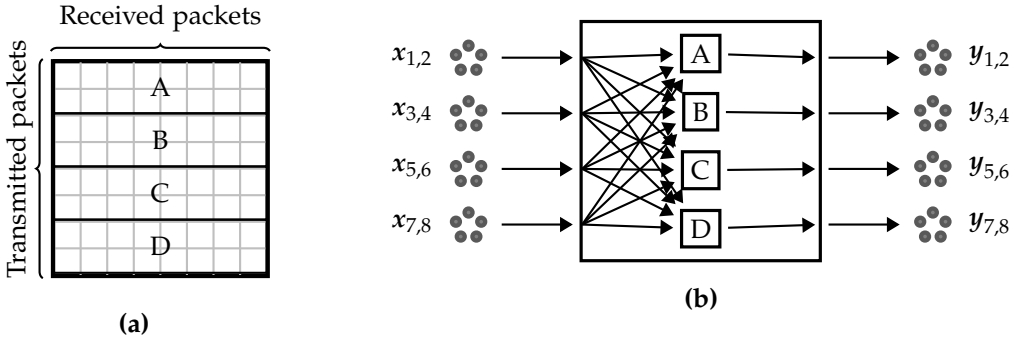


Figure 5.5 – Row-based decomposition of the interposer matrix. The matrix **L** can be partitioned into four smaller matrices (a). In (b) these sub-matrices are mapped to the processing cores which apply them to the vector $\boldsymbol{x}$.

was not used they need now only transmit two *multicast* packets per time step. Likewise, since processor 'A' computes and transmits the values $\sum_{i=1}^{8} \mathbf{L}_{1i}\boldsymbol{x}_i = \boldsymbol{y}_1$ and $\sum_{i=1}^{8} \mathbf{L}_{2i}\boldsymbol{x}_i = \boldsymbol{y}_2$ the processing core simulating the ensemble marked $\boldsymbol{y}_{1,2}$ will only receive two packets per simulation time step (similarly, processing cores simulating other ensembles marked $\boldsymbol{y}$ will receive only two packets per time step). This use of multicast is an effective exploitation of a fundamental SpiNNaker feature – the extra arrows in Figure 5.5 are "free".

Row based decomposition of the matrix, therefore, has ideal traffic characteristics – minimal packets are transmitted and received by the cores connected to the interposer. Unfortunately, while this decomposition works well as long as the matrix has short rows, the size of the matrix which can be allocated to an interposer core is limited by the memory available, the number of packets that can be received and the size of matrix-vector multiplication that can be performed in the time available. Hence as length of the rows increases the number which may be allocated to a processor decreases. Beyond a certain row length the computational cost may outstrip the number of cycles available.

Block partitioning

To prevent the rows of a matrix in a partition from becoming too long, the matrix $\mathbf{L}$ in Figure 5.6 is partitioned into four blocks of size 4×4. Each partition is allocated to a single processing core. These cores can be considered as forming two groups based upon the values they receive: 'A' and 'C', and 'B' and 'D'. Since each group receives the same values the SpiNNaker network can be exploited to multicast the packets transmitted to a group, reducing packet traffic.

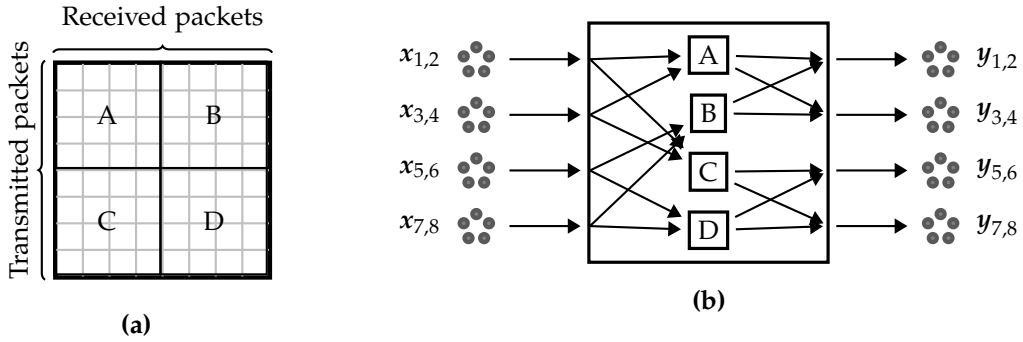


Figure 5.6 – Block-based decomposition of the interposer matrix. By shortening the rows of a decomposed matrix the number of multicast packets received by each processing core in the interposed can be reduced. However, this increases the number of packets received by processing cores connected to the interposer.

The processor marked 'A' computes four product terms, including $\sum_{i=1}^{4} \mathbf{L}_{1i}\boldsymbol{x}_i$. Since it only receives the first four elements of $\boldsymbol{x}$ and contains only the first four elements of the rows of $\mathbf{L}$ it has been allocated, the terms it produces must be added to those produced by the core marked 'B' to form the inputs to the cores simulating the ensembles marked $\boldsymbol{y}$. Among the product terms computed by 'B' is $\sum_{i=5}^{8} \mathbf{L}_{1i}\boldsymbol{x}_i$. Combining these terms results in the first value that must be received by the core simulating the ensemble marked $\boldsymbol{y}_{1,2}$.

$$\sum_{i=1}^{4} \mathbf{L}_{1i}\boldsymbol{x}_i + \sum_{i=5}^{8} \mathbf{L}_{1i}\boldsymbol{x}_i = \sum_{i=1}^{8} \mathbf{L}_{1i}\boldsymbol{x}_i = \boldsymbol{y}_1$$

This addition could be performed in the interposer network – perhaps by using another core – but this would increase interposer latency and consume more processors. Instead the addition can be performed cheaply by the receiving cores, $\boldsymbol{y}$. Consequently, in the network shown in Figure 5.6 each of the ensembles marked $\boldsymbol{x}$ transmits two packets (one per dimension) each time step. These packets are multicast to groups of cores in the interposer. Each core in the interposer network receives four of these packets per time step, performs a matrix vector multiply and transmits a further four packets which are routed to the processors simulating the ensemble $\boldsymbol{y}$ where they are combined to form the inputs for the neurons. These last cores will each receive four packets per time step.

This 'block' form partitioning of the transform $\mathbf{L}$ retains some of the benefits of the 'row' form partitioning described above. Since multiple processing cores receive the same values, the packets containing these values can be multicast using the SpiNNaker network. Not only does this reduce the number of packets which the transmitting cores must send, reducing their computational load, but appropriate physical placement of the interposer cores and routing of the multicast packets can act to reduce the aggregate traffic. This will be illustrated in a later example.

However, since each 'block' corresponds to only some of the columns of the matrix it can produce only a portion of the matrix-vector product. Consequently, processors 'downstream' of the interposer must combine the payloads of multiple packets to form their input vector. A greater number of column partitions means a greater number of packets received 'downstream' of the interposer – see Figure 5.7. Since the number of rows and columns contained within each block can be chosen, there is a trade off between traffic and the computation and memory required for long partitions of the rows.

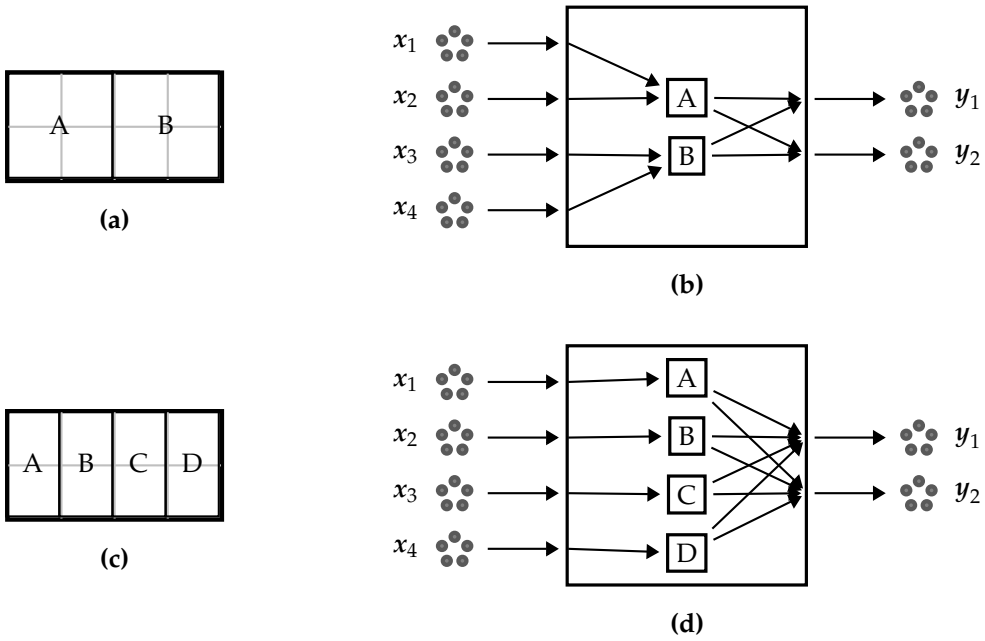


Figure 5.7 – Column-partitions affect the number of packets received downstream. In (a) and (b) the matrix contained with an interposer is partitioned into two groups. This matrix is further divided in (c) and (d) leading to an increase in the number of packets received by y_1 and y_2.

Interposer costs

The costs incurred by an interposer core can be categorised as compute, memory, packet receipt and packet transmission loads as shown in Figure 5.8. While any partitioning of the matrix will reduce the individual memory and compute loads, represented by the area of a partition, only column partitioning can control the number of packets received by each interposer core. However, as seen in Figure 5.7, increasing the number of column partitions increases the load downstream of the interposer. Consequently there is a tension between reducing the number of packets received by each interposer core – achieved through reducing the number of columns allocated to each core – and decreasing the load placed on other cores – achieved by increasing the number of columns in each interposer core. A compromise is required to balance the number of packets received by cores in the interposer with the number of packets which must be received by the downstream cores. Section 5.3 investigates this balance with respect to a neural network implementation of the circular convolution operator.

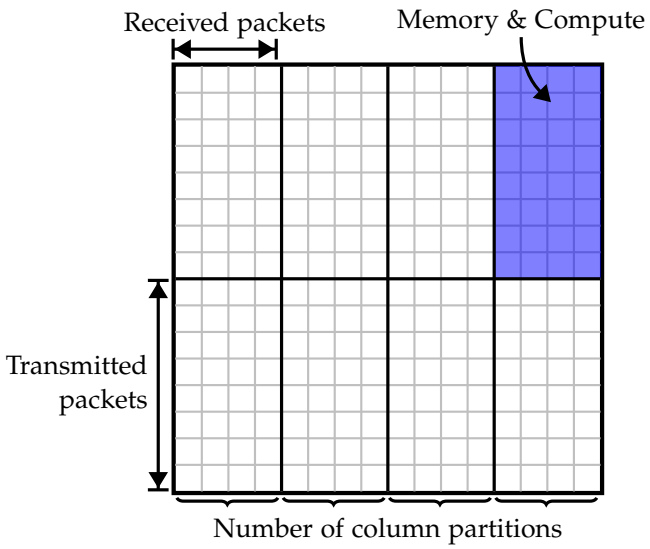


Figure 5.8 – Interposer and partition costs. The columns represent the length of vector that will be received by a partition while the rows represent the length of vector that will be transmitted. Each partition (area) of the matrix must be sufficiently small to meet constraints on memory, compute and received and transmitted packets. However, the number of column partitions must be minimised (to reduce the downstream packet processing costs) as should the total number of partitions (number of processors).

Scheduling and timing

While adding an interposer to a network reduces traffic and the computational load of connected cores it adds an extra stage of processing to simulating a model. Since this processing depends on input from ensembles and produces output for further ensembles it forms part of a pipeline. Determining *when* interposer processing should start is necessary to ensure that each interposer receives all the values it requires before beginning work *and* that those it produces are received in time by further processing stages.

Each interposer is 'surrounded' by ensembles whose simulation is clock driven, this being a real time constraint. A core simulating an ensemble uses a timer to trigger the neural processing and decoding stages described in Section 4.2. To ensure neural simulation occurs in step, these timers are synchronised across the SpiNNaker machine. Consequently, simulation of a neural model consists of periods of computation interspersed with periods of network activity. Interposer processing, which can only occur once all incoming values have been received and needs to be completed fast enough to allow outgoing values to be delivered before the next tick, must be included in these interleaved periods of computation and communication. There are a few ways this may be achieved.

Firstly, each interposer could wait until it received its full input set before starting its processing. Unfortunately, as the SpiNNaker network does not guarantee packet delivery interposers could become 'stuck' waiting. As the cores surrounding an interposer are clock driven they would continue processing irrespective of the fault and since they would receive no packets from the interposer the ensembles they simulate would receive no input. Consequently, the loss, or delay, of even one packet, leading to a 'stuck' interposer, could seriously affect the accuracy of the simulation.

Since *asynchronous* interposers could lead to simulation inaccuracy a clock should be used to ensure that interposer processing is triggered even in the presence of network faults. This clock could be used in one of two ways: interposers could be entirely *synchronous* or they could wait until either their complete input set arrives or the clock ticks (whichever is sooner). Regardless of the method chosen this second clock must have the same period as that which drives ensemble processing, but lag in phase. This phase difference should be chosen to allow enough time for the inputs to the interposers to become available and could be different for each interposer. Figure 5.9 illustrates these two clocks and the phase difference required.

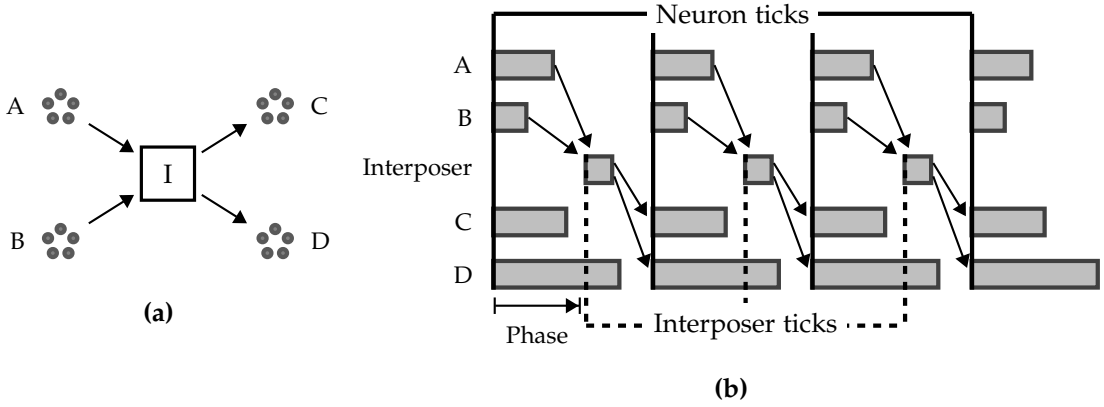


Figure 5.9 – Interposer timing. A neural network is shown in (a). In (b) the clock driven computation and communication used to simulate the network are illustrated.

There are now two constraints upon the period of the simulation clock: the ensemble simulation time and the interposer processing time. Specifically, the compute time allowed for a step of the simulation must be at least as long as the time required by the slowest pairing of ensemble and interposer. As this may be longer than the time required by the slowest ensemble alone it may not be possible to run as fast as biology, offsetting some of the gains of the interposers. Fortunately, since SpiNNaker is a "processor

rich" environment, these times can be reduced by partitioning ensembles and interposers across more parallel processors. For example, using more cores to simulate an ensemble reduces the number of neurons which must be simulated by each core and thus reduces the time required to simulate the ensemble. Likewise, using more interposer cores, and allocating fewer rows to each core, reduces the time required by the interposer. How these times compare to the 1 ms time step allowed for biological real time simulation of a core component of Spaun will be seen in Section 5.3.

It should be noted that the tasks of simulating ensembles and performing interposer processing are unlikely to overlap. This implies that the same cores could be used to perform both tasks. Since this would place even more demand on the limited amount of memory available to a processor this idea has not been pursued further. As SpiNNaker has many processors profligacy is no handicap and, as Sections 5.3 and 5.3 demonstrate, interposer costs are small relative to their gains.

Summary

The high dimensional spaces required to ensure that the vectors (symbols) in the Semantic Pointer Architecture remain distinct necessitate the use of ensembles containing many neurons. Since choosing decoders for ensembles of this nature is prohibitively expensive the high dimensional spaces are instead spread over a number of smaller ensembles. Connecting these groups of ensembles such that the values they transmit are transformed increases the density of network connectivity and the number of packets that must be received by processing cores.

To overcome the problems associated with dense network connectivity and the requirement that large numbers of packets be received by processing cores a non-neural 'interposer' was introduced. By moving the application of a matrix-vector multiplication from the processing cores simulating ensembles to processing cores dedicated to the task, both the density of network traffic and the number of packets received by other processing cores can be decreased. In addition, fewer packets must be transmitted by cores 'upstream' of the interposer and those that are can, unlike before, be multicast – exploiting a feature of the SpiNNaker network to reduce a number of costs.

5.3 Circular convolution

Section 5.2 introduced an interposer to reduce the network congestion and compute load resulting from densely interconnected neural networks. Similarly dense connectivity is found, albeit in a much larger scale, in the neurally-inspired implementation of the circular convolution operator (Section 2.3) used in Spaun. In this section the principles explored in Section 5.2 are put into practice to reduce the loads placed on a SpiNNaker machine when simulating this core part of the Spaun model.

The Circular Convolution of two vectors, $\boldsymbol{a} \circledast \boldsymbol{b}$, yields a new vector which is similar to neither of the original vectors but contains sufficient information about them that either may be recovered given the other (see Section 2.3). Circular convolution can be expressed as $\boldsymbol{a} \circledast \boldsymbol{b} = \mathcal{F}^{-1}\left(\mathcal{F}\boldsymbol{a} \odot \mathcal{F}\boldsymbol{b}\right)$ where $\mathcal{F}$ is the Discrete Fourier Transform and $\odot$ is element-wise multiplication $\boldsymbol{x} \odot \boldsymbol{y} = \begin{pmatrix} x_1 y_1 & x_2 y_2 & \cdots & x_{n-1} y_{n-1} & x_n y_n \end{pmatrix}$ (Plate 1995).

Figure 5.10 illustrates the neural network which implements circular convolution in Spaun. The vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ are 512-dimensional. To reduce the time required to choose decoders for the ensembles representing these values a number of smaller ensembles are used, as described before. Consequently, the groups of ensembles labelled $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ each consist of thirty-two 800-neuron ensembles each representing 16-dimensions.

To compute the circular convolution of $\boldsymbol{a}$ and $\boldsymbol{b}$ they must be transformed into the Discrete Fourier Domain. Therefore, the connections from $\boldsymbol{a}$ and $\boldsymbol{b}$ to the groups of ensembles in the 'Product' network apply the $\mathcal{F}$ matrix to their input values. Since the element-wise multiplication of $\mathcal{F}\boldsymbol{a}$ and $\mathcal{F}\boldsymbol{b}$ is required some additional factors are added to the connections (Gosmann 2015) such that 'sq1' receives $\frac{1}{\sqrt{2}}\mathcal{F}\boldsymbol{a}$ and $\frac{1}{\sqrt{2}}\mathcal{F}\boldsymbol{b}$ as input whereas 'sq2' receives $\frac{1}{\sqrt{2}}\mathcal{F}\boldsymbol{a}$ and $-\frac{1}{\sqrt{2}}\mathcal{F}\boldsymbol{b}$. The groups of ensembles labelled 'sq1' and 'sq2' each contain one thousand and twenty eight scalar-representing 100-neuron ensembles. The decoders of these ensembles are chosen to compute the square of the values they represent (see Section 2.2, p. 40). These decoded values are transformed by the inverse Discrete Fourier Transform, $\mathcal{F}^{-1}$, and additional factors which complete the element-wise multiplication: $\frac{1}{2}$ in the case of 'sq1' and $-\frac{1}{2}$ for 'sq2'. Consequently the ensembles in $\boldsymbol{c}$ receive the value:

$$\begin{aligned}\boldsymbol{c} &= \frac{1}{2}\mathcal{F}^{-1}\mathrm{sq1}^2 - \frac{1}{2}\mathcal{F}^{-1}\mathrm{sq2}^2 \\ &= \frac{1}{2}\mathcal{F}^{-1}\left(\frac{1}{\sqrt{2}}\mathcal{F}\boldsymbol{a} + \frac{1}{\sqrt{2}}\mathcal{F}\boldsymbol{b}\right)^2 - \frac{1}{2}\mathcal{F}^{-1}\left(\frac{1}{\sqrt{2}}\mathcal{F}\boldsymbol{a} - \frac{1}{\sqrt{2}}\mathcal{F}\boldsymbol{b}\right)^2 = \mathcal{F}^{-1}\left(\mathcal{F}\boldsymbol{a} \odot \mathcal{F}\boldsymbol{b}\right)\end{aligned}$$

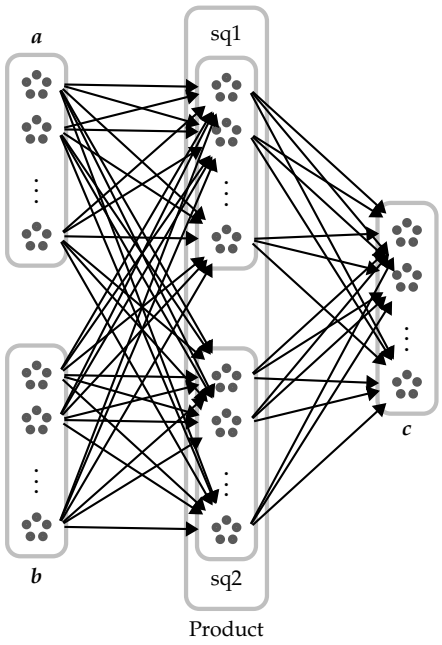


Figure 5.10 – Neural network implementation of circular convolution. Each of the 512-dimensional values $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ are represented by a group of thirty-two 800-neuron ensembles, each representing a 16-dimensional value.

As is apparent in Figure 5.10 the network connectivity is exceptionally dense. Table 5.1 details the number of packets that processors simulating ensembles in different portions of the network would need to transmit or receive every time step. In the worst case a core simulating an ensemble in the part of the network labelled $\boldsymbol{c}$ would need to dedicate 80 % of its cycles to the task of processing received packets. Consequently, to meet compute requirements (see Section 4.4), six cores would be required to simulate each of the thirty-two ensembles in $\boldsymbol{c}$ – a total of 192 cores.

	Packets per time step per ensemble	
Portion of network	Transmit	Receive
$\boldsymbol{a}$	1028	
$\boldsymbol{b}$	2056	
sq1 or sq2	512	64
$\boldsymbol{c}$		2056

Table 5.1 – Network utilisation of circular convolution. Rows describe the number of packets received and transmitted by a cores simulating one of the ensembles in a region of the network. Packets transmitted by processing cores simulating ensembles in $\boldsymbol{a}$, *but not those in* $\boldsymbol{b}$, can be multicast to those simulating ensembles in 'sq1' and 'sq2'.

Interposer parameter selection

Figure 5.11 shows the addition of interposers to the circular convolution network. Since the matrices contained within each interposer are large, decomposition of the interposers, as described above, is required.

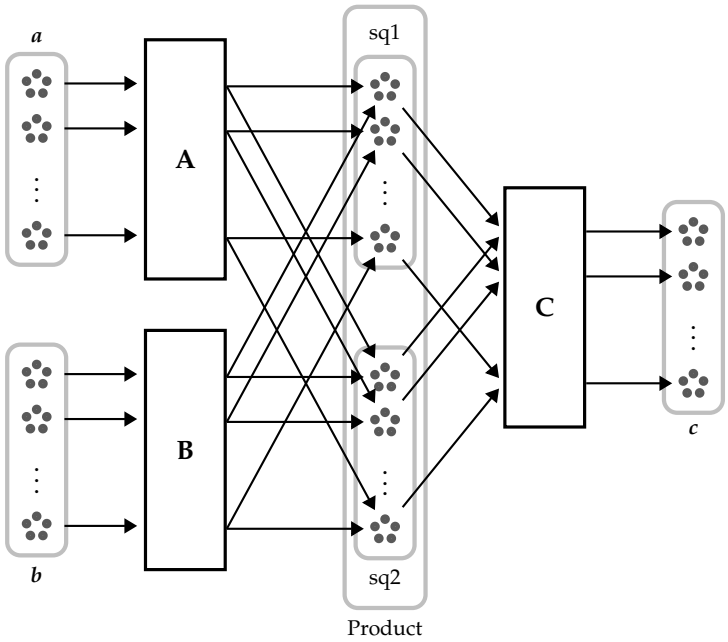


Figure 5.11 – Circular convolution network with interposers. Interposer **A** contains a 1028×512 matrix, **B** includes a 2056×512 matrix and **C** a 512×1028 matrix.

To assess the effect of different decompositions of the interposers upon usage of the SpiNNaker network an experiment was performed. A number of maximum partition sizes which met the memory, compute and traffic constraints described in Section 5.2 (p. 107) were chosen. For each partition size a model of the circular convolution network was constructed. Each of these models was placed and routed onto a representation of a physical 288-chip SpiNNaker machine and the number of packets that would traverse each link every simulation time step was computed. The networks were placed using a simulated annealing strategy (Heathcote 2016, Chapter 6) and routed using Neighbourhood Exploring Routing (Navaridas et al. 2015).

Figure 5.12 shows the distribution of the network loads on active links in the SpiNNaker machine for these decompositions of the network. It should be noted that none of the interposer partitioning schemes result in traffic greater than 20 % of the maximum bandwidth of a SpiNNaker link (1 Gibit s^{-1}). Consequently real time SpiNNaker simula-

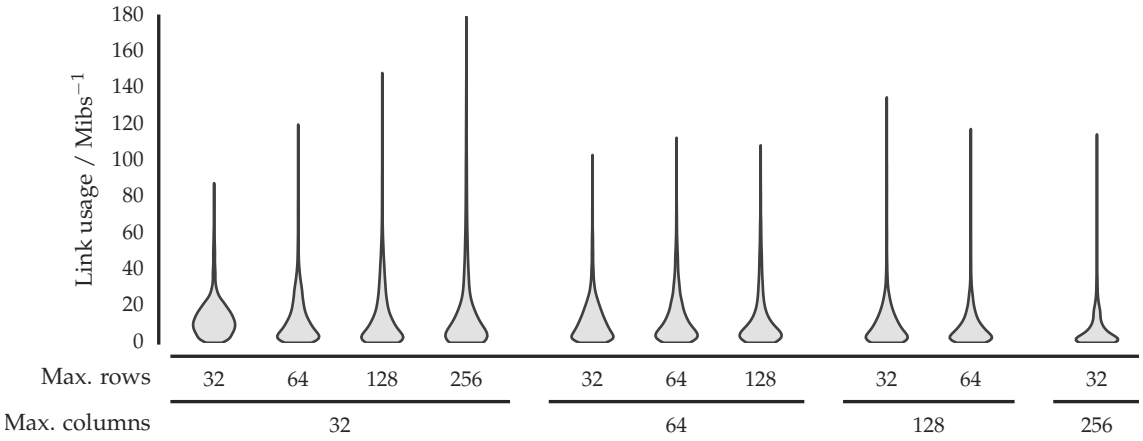


Figure 5.12 – Distribution of loads on SpiNNaker links for various decompositions of the circular convolution network. Each 'violin' is a vertical histogram showing the proportion of active links carrying a given load for different maximum block sizes.

tion of the circular convolution network is feasible when interposers are used, subject to packets being sufficiently dispersed in time.

Table 5.2 shows the number of packets per time step arriving at processing cores simulating ensembles in 'sq1', 'sq2' or ***c***. Firstly, it is apparent that *any* interposer partitioning leads to a reduction in the number of packets arriving at processing cores in ***c***. It is desirable that the load placed on the interposer be balanced with the load presented downstream of the interposer. This would suggest that partitions should contain a maximum of 128 columns.

Maximum column size	Received packets per time step per ensemble sq1 or sq2	***c***
32	32	528
64	16	272
128	8	144
256	4	80
No interposer	64	2056

Table 5.2 – The number of packets received every time step by ensembles in different areas of the network can be reduced by inclusion of an interposer.

Two of the schemes presented in Figure 5.12 are suitable and, since the loads they place on the SpiNNaker network are similar, selecting the one uses the fewer cores is reasonable. This would suggest that 64×128 is the maximum interposer partition size for the circular convolution network. Given this, the number of processing cores required in the SpiNNaker instantiation of this network is as shown in Table 5.3.

Portion of network	Cores required	
	No interposer	64×128 interposer
a	96	96
A		68
b	96	96
B		136
'sq1' and 'sq2'	2056	2056
C		72
c	192	96
Total	2440	2620

Table 5.3 – Cores required in SpiNNaker instantiation of circular convolution

Inclusion of the interposer increases the number of processing cores required to simulate the circular convolution network. However, this is more than offset by the reduction in the number of packets received (Table 5.2) and transmitted (Table 5.4) by processing cores in the network.

Portion of network	Packets transmitted per time step per ensemble	
	Without interposer	With interposer
a	1028	16
b	2056	16
sq1 or sq2	512	1

Table 5.4 – The number of packets transmitted each time step by ensembles in different areas of the network is reduced if an interposer is included.

Additionally, since the packets transmitted upstream of an interposer can be multicast the resulting reduction in aggregate traffic is significant. This is shown in Figure 5.13 which compares the expected link traffic of the circular convolution network with no interposer with that which results when interposers consisting of 64×128 blocks are included.

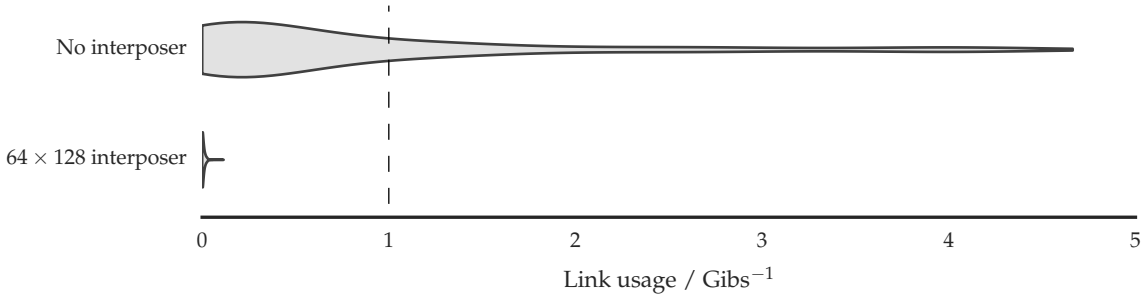


Figure 5.13 – Circular convolution link usage with and without interposers.

When interposers are not included in the circular convolution network the link bandwidth of 1 Gibit s^{-1} is exceeded. To avoid innaccuracy resulting from network congestion the simulation would need to proceed around five times slower than biological real time. In contrast, the reduction in traffic resulting from the use of interposers means that real time simulation is possible.

This reduction in traffic is a consequence both of fewer packets being transmitted *but* also the exploitation of the multicast capability of the SpiNNaker network: allowing packets to be duplicated *as late as possible*. Figure 5.14, which shows a portion of the place and route solution for the circular convolution network with interposers of size 64×128, illustrates this principle. The single net shown connects one ensemble in $\boldsymbol{a}$ to the interposer. Despite the net connecting to 17 processors spread across eight chips there are few parallel connections between chips. Instead, since the interposer cores have been placed together, there are only seven places where the net forks and, of these, only two forks cause packets to be transmitted across multiple links.

Timing

One of the consequences of including an interposer in a network is that stages of neural and interposer processing must be interleaved (Section 5.2). Since this can increase the time required to simulate a step of the network it is worth investigating whether real time simulation of the circular convolution net with an interposer is possible.

The interposer scheme suggested in Figure 5.11 introduces interposers between the ensembles $\boldsymbol{a}$, $\boldsymbol{b}$ and 'sq1' and 'sq2', and between 'sq1', 'sq2' and $\boldsymbol{c}$. Simulation in biological time requires that no more than 1 ms is taken to simulate these ensembles *and* perform the interposer processing. The number of cycles needed to simulate an ensemble is given by Eq. 4.4 (p. 87). Each ensemble in $\boldsymbol{a}$ and $\boldsymbol{b}$ contains 800 neurons and represents 16 dimensions but, to meet memory constraints, was partitioned over three cores. Consequently, each step of these ensembles consumes 330 µs. Ensembles in 'sq1' and 'sq2' each use 100 neurons to represent a scalar, hence simulation of each step requires 40 µs; since this is less than that required by $\boldsymbol{a}$ and $\boldsymbol{b}$ it can be disregarded. The processing of a 64×128 interposer block was found to require around 250 µs. As $330\,\mu s + 250\,\mu s < 1\,ms$, simulation in biological real time can be achieved – even allowing a reasonable margin of error for delays in packet delivery.

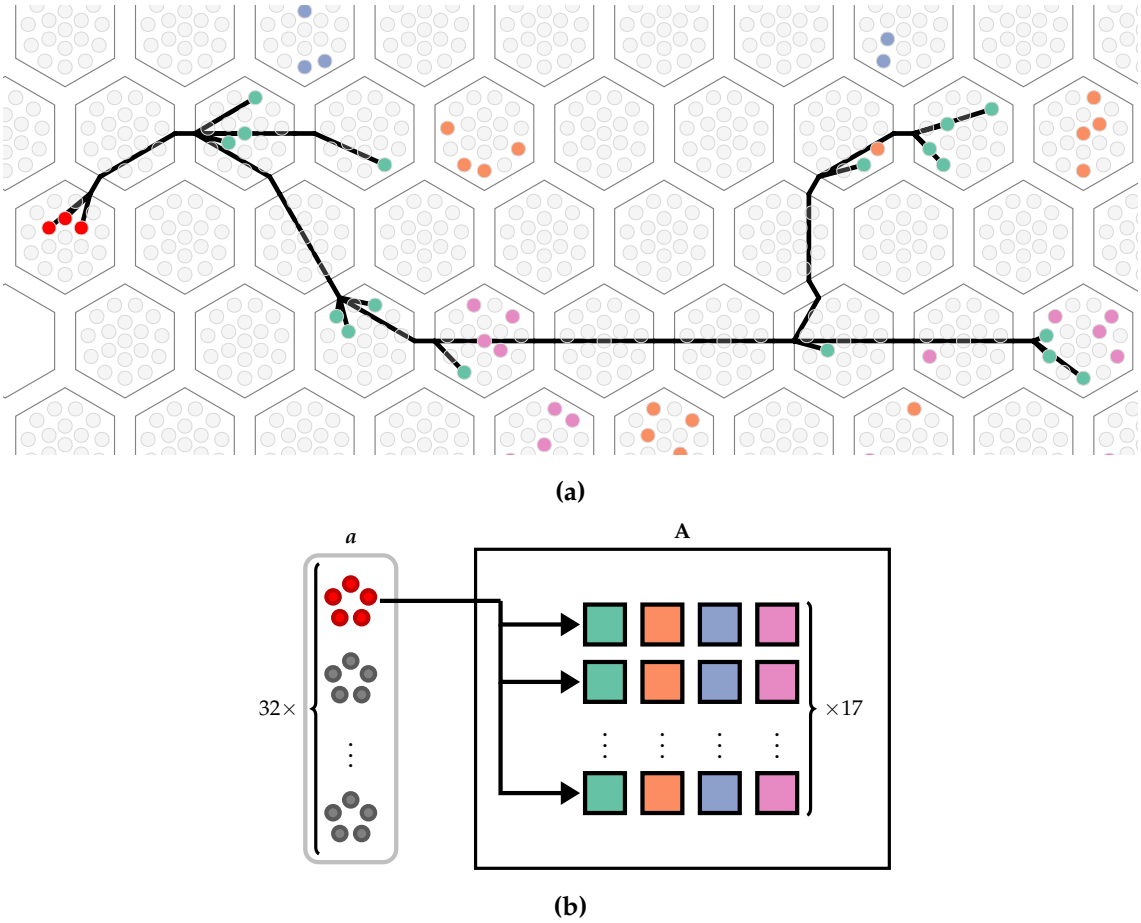


Figure 5.14 – Section of the place-and-route solution for circular convolution using a maximum block size of 64 × 128. The place-and-route solution is shown in (a). Each hexagon represents a SpiNNaker chip and each circle represents a core which is in use. The red cores at the left of the image are being used to simulate an 800-neuron ensemble in $\boldsymbol{a}$. The other coloured cores represent the cores simulating interposer A, equivalently coloured cores contain different rows of the *same columns* of the interposer. One net connecting the ensembles (red) to the inputs of one column division of A (green) is shown. The highlighted components and single net are illustrated in (b).

Summary

The interposer technique presented in this chapter enables real time simulation of a component of Spaun which could not otherwise be simulated in real time using the simulation scheme introduced in the previous chapter. However, since the main costs associated with the circular convolution net are due to the high degrees of network fan-out and the SpiNNaker network is optimised to multicast spiking traffic with high fan-out, it is worth considering whether spike-, rather than value-based transmission, would lead to more efficient use of the architecture for this network.

Comparison to spiking implementation

The previous chapter identified that the two main constraints on spike-based simulation of neural networks constructed using the Neural Engineering Framework (NEF) were the needs to store large, dense synaptic weight matrices and to handle the many synaptic events that resulted from high-firing rates and dense weight matrices.

To aid in determining whether spike-transmission would result in more efficient simulation of the circular convolution network an experiment was performed to measure the rate at which spikes arrived at different parts of the network. These *afferent* spike rates are summarised in Table 5.5 along with the number of synaptic events and the memory consumed by the weight matrices for each ensemble.

Sharp and Furber (2013) found that a processing core could support a maximum of 5000 synaptic events per millisecond time step when running in biological real time. This suggests that an ensemble in $\boldsymbol{c}$ would need to be partitioned over $\frac{14\times10^9}{5\times10^6} = 2800$ cores. Since such an ensemble contains only 800 neurons this is not possible, but simulation *could* occur at one quarter of biological real time if 800 cores, simulating one neuron each, were used. At this rate of simulation each of the 2056 ensembles in 'sq1' and 'sq2' would need to be partitioned across 20 processing cores. Consequently, a total of 66 720 cores would be required to simulate the ensembles in 'sq1', 'sq2' and $\boldsymbol{c}$. If the simulation were

		Per ensemble	
Portion of network	Afferent spikes / 10^6 s^{-1}	Synaptic events / 10^9 s^{-1}	Weight matrices / MiB
'sq1' and 'sq2'	4.3	0.4	9.8
$\boldsymbol{c}$	17.5	14.0	313.7

Table 5.5 – Costs of spike-transmission for the circular convolution network

to be run yet slower fewer cores would be required.

Since the above analysis has disregarded the traffic and the number of cores required to simulate ensembles in $\boldsymbol{a}$ and $\boldsymbol{b}$, a greater number of cores than stated would be required and the maximum rate of simulation may be slower. Even allowing for these inaccuracies, the value-based simulation with interposers requires 96 % fewer cores to simulate the circular convolution network at $4\times$ the speed of the spike-based method. While there are optimisations that could be used to reduce the number of cores, or increase the simulation rate, of the spike-based transmission they are unlikely to achieve gains similar to those of the value-based technique introduced in this thesis.

5.4 Results

To assess the accuracy of the circular convolution implementation on SpiNNaker a number of experiments were run. Fifty instances of the circular convolution model were simulated on SpiNNaker (using the interposer technique described in this chapter) and on a standard PC using Nengo. The instances were seeded equivalently so that accuracy could be compared across the simulators. Two measures were used to assess performance: the magnitude of the convolved vector produced by the simulation and the dot product between the simulated and expected convolved vectors. Figure 5.15 illustrates how these measures changed over time for a single 500 ms simulation and Figure 5.16 shows the correlation between the Nengo and SpiNNaker results for all fifty simulations.

It is apparent that, although the results are strongly correlated across the platforms, the vectors decoded from the SpiNNaker simulations are both shorter (of lower magnitude) and less like the expected vectors than those decoded from the Nengo simulations (the dot product between the simulated and expected vectors are smaller). These discrepancies likely result from inaccuracies caused by the use of fixed point representations on SpiNNaker. The extent to which these differences affect the correctness of simulations is highly context dependent.

For example, in Spaun the dot products between the neural representation of a vector and a vocabulary of other vectors are used to determine which actions should be taken. An instance of this was shown in Section 2.4. Consequently, one measure of whether the errors introduced through SpiNNaker simulation would affect the *behavioural* performance of model is to investigate the likelihood that the neurally computed convolution

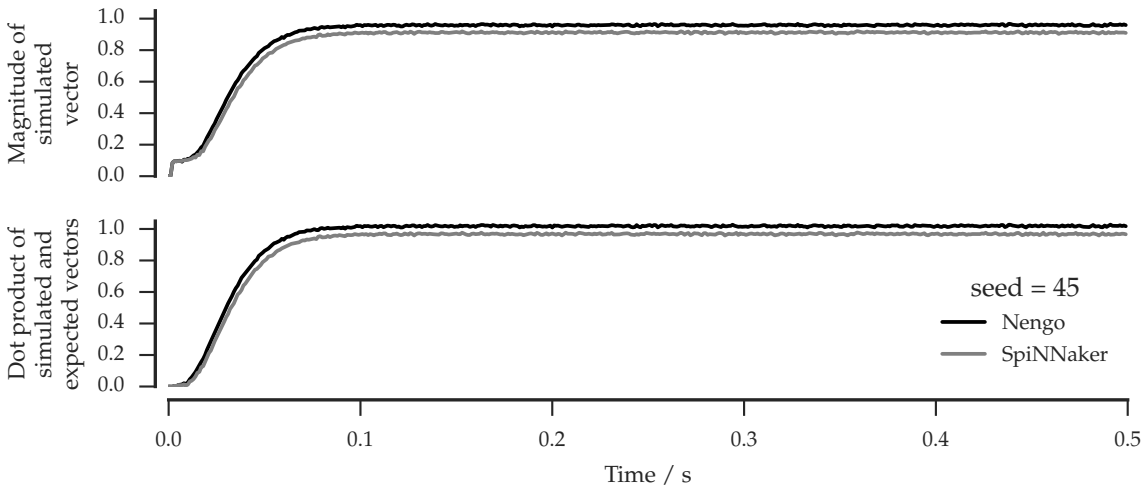


Figure 5.15 – Sample of 512-D circular convolution simulation results. The upper plot demonstrates how the magnitude of the vector output by an instantiation of the circular convolution network (with seed $= 45$) varies over time when simulated both on SpiNNaker and with Nengo. The lower plot shows the dot product between the expected and simulated vector for the same network instance, the closer to 1 the more similar the vectors and the more correct the output of the network.

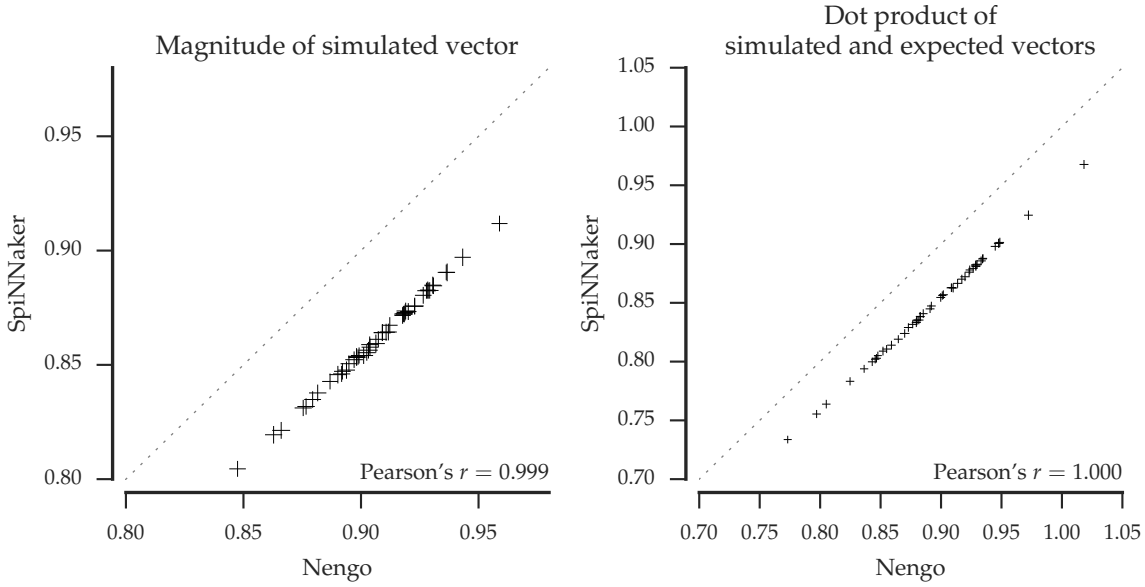


Figure 5.16 – Comparison of SpiNNaker and Nengo 512-D circular convolution simulation results. Each of the fifty points represents one instance of the circular convolution network (with seeds 0 to 49). The left panel shows how the steady state mean of the magnitude of the output vector (as measured from 0.1 s to 0.5 s – see Figure 5.15) corresponds between Nengo and SpiNNaker. The right panel shows how the steady state mean of the dot product of the output and correct vectors correspond across the two platforms. Error bars show the standard deviation of the values.

of a pair of vectors, $\widehat{\boldsymbol{a} \circledast \boldsymbol{b}}$, is more similar to the convolution of another pair of vectors, $\boldsymbol{c} \circledast \boldsymbol{d}$, than to the canonical version of itself, $\boldsymbol{a} \circledast b$:

$$p\left(\widehat{\boldsymbol{a} \circledast \boldsymbol{b}} \cdot \boldsymbol{c} \circledast \boldsymbol{d} - \widehat{\boldsymbol{a} \circledast \boldsymbol{b}} \cdot \boldsymbol{a} \circledast \boldsymbol{b} \geq 0\right) \tag{5.1}$$

In Figure 5.17 the two terms of Equation 5.1 are shown separately before being combined. It is clear that although the inaccuracies present in SpiNNaker are sufficient to slightly shift the mean of the final distribution (shown on the left) closer to the origin there remains a vanishingly small likelihood that one convolved vector could be mistaken for another. This is not unexpected given the incredibly low likelihood of randomly selecting two similar vectors from the surface of a 512-D hypersphere. Consequently, we should expect that any neural system requiring this form of vector comparison should behave similarly regardless of whether it is simulated on Nengo or SpiNNaker. Necessarily,

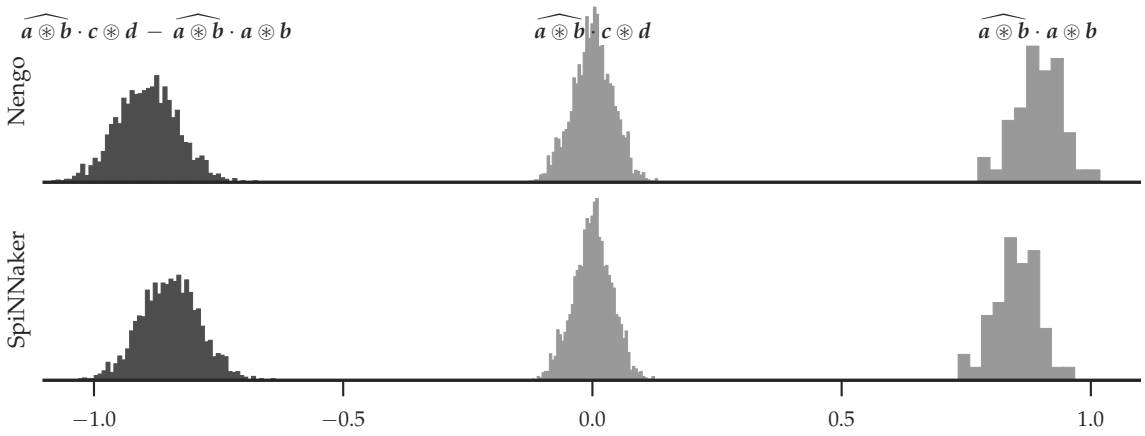


Figure 5.17 – Distributions over the dot products of convolved vectors produced by neural simulations on Nengo and SpiNNaker. Each axis shows three distributions. On the right is the distribution over the similarity between a convolved vector produced by a neural simulation and the correct version of that convolved vector (50 samples). The centre distribution represents the similarity between a convolved vector produced by a neural simulation and a randomly selected unit-length 512-D vector ($50 \times (50 - 1) = 2450$ samples). Finally, the distribution on the left can be used to derive the likelihood that a randomly selected unit-length vector is *more* similar to a neurally produced convolved vector than the neurally produced convolved vector is to the correct version of itself (2450 samples). Note that while the means of the two outer distributions are shifted closer to 0 for SpiNNaker when compared to Nengo it remains vanishingly unlikely that a value drawn from the leftmost distribution will be greater than or equal to zero and hence that a convolved vector could be mistaken for another.

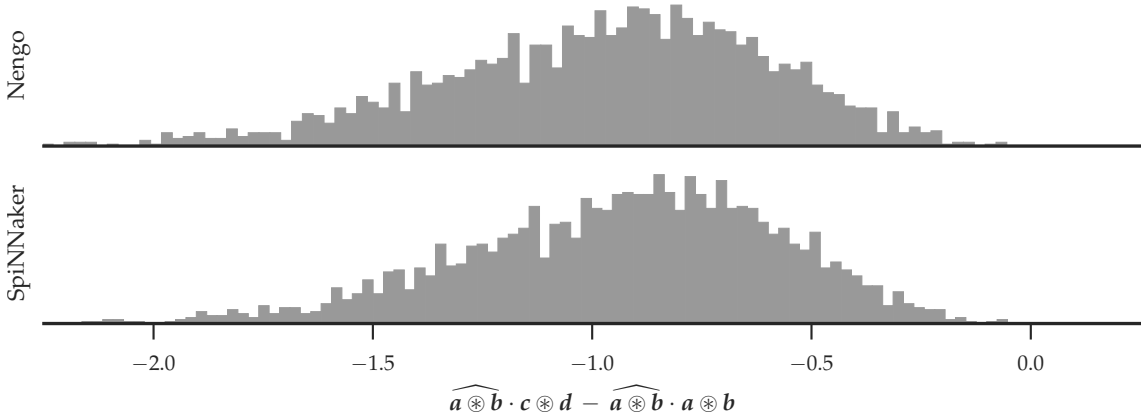


Figure 5.18 – Distribution over the dot products of convolved 16-D vectors (2450 samples). The distributions pictured should be compared to the leftmost distributions of Figure 5.17. When using lower dimensional vectors, as shown here, the spread of the distribution pictured is considerably greater than when higher dimensional vectors are used. A result of this is that, although it remains unlikely that a convolved vector could be confused for another convolved vector ($p\left(\widehat{\boldsymbol{a} \circledast \boldsymbol{b}} \cdot \boldsymbol{c} \circledast \boldsymbol{d} - \widehat{\boldsymbol{a} \circledast \boldsymbol{b}} \cdot \boldsymbol{a} \circledast \boldsymbol{b} \geq 0\right)$ is small) it is more likely than when higher dimensional vectors were used.

this analysis depends on the vectors being selected at random from a high dimensional space. For example, if a lower dimensional space were used (e.g., 16 dimensions as in Figure 5.18) then the likelihood of vectors being mistaken increases. Likewise, if vectors were not selected randomly but were instead clustered such that related concepts were positioned more closely within the space (e.g., the dot products between vectors representing colours were consistently greater than between those representing, say, *red* and *calculus*) then it may be more likely that vectors are confused *within* a category (e.g., the vector representing *red* being confused with the vector representing *rose*).

Figures 5.16 and 5.17 suggest that while the vectors resulting from SpiNNaker simulation of the circular convolution network are not equivalent to those resulting from Nengo they are sufficiently similar that the two systems will produce equivalent behaviour. However, it is suggested that further experiments are performed to determine cases in which differing behaviour occurs. In particular, one might expect the performance of a SpiNNaker simulation of a model performing repeated convolution of the same vectors to become dominated by noise faster than would be expected of Nengo simulation of the same network. SpiNNaker performance might be improved by varying the fixed point representation used to store encoders and decoders or by including hardware support for floating point numbers in any future chip.

5.5 Summary

The high dimensional spaces required by the Semantic Pointer Architecture place heavy demands on SpiNNaker. Specifically, the dense connectivity between ensembles this entails requires much network traffic, increasing the load upon both the network and the receiving cores and thus the likelihood of packets being dropped or real time deadlines missed. These problems, which would reduce the accuracy of the simulation, could be mitigated by decreasing either the simulation rate or granularity. However, a third option was proposed: an 'interposer' was introduced which, by exploiting the *multicast* capability of the network and nature of the communication, reduced the traffic.

The utility of this interposer was studied with respect to the neural network which implements the circular convolution operator in Spaun. Table 5.6 summarises the time and SpiNNaker resources required to simulate this network using the methods discussed in this thesis.

Method		Cores required	Percentage of real time / %
Spike-based	(estimated)	66 720	25
Value-based	without interposers	2440	20
	with interposers	2620	100

Table 5.6 – Comparison of simulation methods for Circular Convolution network. Figures shown are for the fastest possible simulation or biological real time.

Packets transmitted across the SpiNNaker network are routed according to the entries contained in routing tables. High degrees of fan-in and fan-out tend to result in these tables becoming too large to store in hardware; of the 213 routing tables generated by one placement of the circular convolution network, eight were too big. The following chapter presents methods for reducing the size of these tables.

Chapter 6

Routing table minimisation

Previous chapters discussed the problems facing the simulation of large neural nets on the SpiNNaker architecture. Chapter 4 provided solutions that allowed efficient simulation of the Neural Engineering Framework on the platform, despite the limited memory and compute resource afforded to each node. Chapter 5 investigated the challenges posed by features of the Semantic Pointer Architecture to the compute resources of processing cores and to the network fabric. As networks simulated on SpiNNaker grow in both scale and complexity they are likely to begin to impinge on yet another resource in the system: the routing table. This chapter focusses on making better use of routing table entries through logic minimisation techniques. It largely reproduces a publication by the author (Mundy, Heathcote, and Garside 2016). While the circular convolution network, and similar networks, can sometimes result in routing tables which are too large this chapter uses example networks inspired by the intended use of SpiNNaker.

6.1 Introduction

SpiNNaker cores communicate by transmitting short packets consisting of an 8 bit header, a 32 bit *key* and an optional 32 bit payload. These packets traverse the links in the SpiNNaker network between routers where they are forwarded, and possibly duplicated, to further links and processing cores.

At each router packets are recognised using a Ternary Content Addressable Memory (TCAM) which bit-masks the packet keys before looking for particular matches, this means that each key bit can be matched with binary `0`, `1`, or `X` ('don't care'). The TCAM

is prioritised such that only the first-found match will be returned, allowing 'catch all' entries to be inserted near the bottom of the table. If a key is not found in the table, the associated packet is 'default-routed': continuing in the direction it arrived.

For example, given a simplified routing table with 4-bit keys:

Key-Mask	Route
`0000`	`NE N`
`X111`	`S`
`1XXX`	`3 4`

packets with the key `0000` would match only the first entry and would be routed out of both the North East and North links. Any packets with the keys `0111` or `1111` would match the second entry in the table and be routed out of the South link. Other packets beginning with a `1` would match the last entry and be routed to the indicated processing cores on the same chip as the router. All other packets would be unrecognised and be default-routed in a straight line, e.g., a packet arriving on the South West link with the key `0011` would be routed out of the North East link.

The TCAM was chosen with an arbitrary size of 1024 entries, believed large enough for most applications although making entries quite precious. Indeed, it is not always possible to avoid exceeding this limit on routing table size – particularly for neural networks which feature large fan-in or fan-out routes, such as the circular convolution network illustrated in the previous chapter. There are several ways to reduce the size of the routing tables. Firstly, the place and route strategies, which allocate applications to the processors in the SpiNNaker network and determine the paths taken by packets, can be constructed to reduce the number of entries required. For example, the Neighbourhood Exploring Routing algorithm (Navaridas et al. 2015) and the simulated annealing placement used in the previous chapter (Heathcote 2016) both result in fewer routing table entries than other place and route techniques. However, if tables are still too large then they can be minimised using logic minimisation.

6.2 Benchmarks

Two benchmark networks were constructed to indicate the importance of routing table minimisation on SpiNNaker. In the first of these, each processing core in a 12×12, 144-chip (2592-core) system was connected, with a probability determined by the distance,

to every other core in the system. This models networks in which processing cores are strongly connected to their neighbours but are weakly connected to distant chips: similar forms of distance-dependent connectivity are found in some brain regions (e.g., Hellwig 2000; Bassett et al. 2010).

The second benchmark extends the first by adding a number of longer distance connections to the network and is based on the 'centroid' model of Navaridas et al. (2015). In this model a small number of cores transmit packets to additional groups of cores located at some distance across the machine. Figure 6.1 illustrates these two benchmarks and shows the probability of a single core being connected to other cores within the network.

Figure 6.1 – Map of the probability of a single core being connected to cores on surrounding chips for the two benchmarks. Darker means higher probability. In the above centroid model the core is connected to two clusters of cores (toward the South West and North West).

Routing keys were assigned such that packets originating on the pth core of the chip at (x, y) in the skewed mesh used the two most significant bytes in the 32-bit key to represent the x and y co-ordinates and the following five bits to indicate the core ID, p. The remaining 11 bits of the key were left blank, i.e., `x` in the routing tables. This 'XYP' key assignment scheme is common in SpiNNaker applications (Davies et al. 2012).

Once the connectivity of the benchmarks was determined, the Neighbour Exploring Routing (NER) algorithm (Navaridas et al. 2015) was used to generate the routes taken by packets across the network. While this algorithm has been shown to exploit 'default routing' to reduce the size of routing tables, the routing tables of both benchmarks were too large to fit in the number of entries available, even when entries which could be replaced by default routing were removed. Figure 6.2 shows both the original size of the routing tables and the sizes once default routes were removed.

6.3 Routing table compaction

Liu (2002) presented a method for using Espresso-II (Brayton 1984) to minimise routing tables for IP routers. In this method a routing table is partitioned in subtables containing entries with equivalent destinations and prefix length. Each subtable is minimised using Espresso to produce a functionally equivalent but smaller set of entries. Figure 6.2 shows the result of applying this technique to the routing tables from our benchmarks, once entries which could be replaced by default routing were removed. In the case of the 'locally-connected' benchmark a combination of default routing and logic minimisation is able to reduce the majority, but not all, of the 144 routing tables to fewer than 1024 entries. In the more challenging centroid model this technique is unable to reduce *any* tables to fit within the TCAM constraint.

In contrast to the unicast IP routing tables, at which minimisation is normally targeted, SpiNNaker routing tables are multicast. Consequently, there are two factors which may explain the poor compression ratios achieved by the above technique when applied to SpiNNaker routing tables. Firstly, while the IP routing tables discussed by Liu (2002) have relatively few output routes (e.g., tens), each SpiNNaker router has the equivalent of up to 2^{24} unique routes (any combination of six inter-chip links and 18 processors) – consequently the number of entries with equivalent routes can be expected to be much smaller. Secondly, whereas IP addresses are assigned such that coarse routing decisions may be made from few bits, multicasting potentially reduces the amount of mutual information between keys which describe similar routes since keys are typically related to route *origins*, not *destinations*.

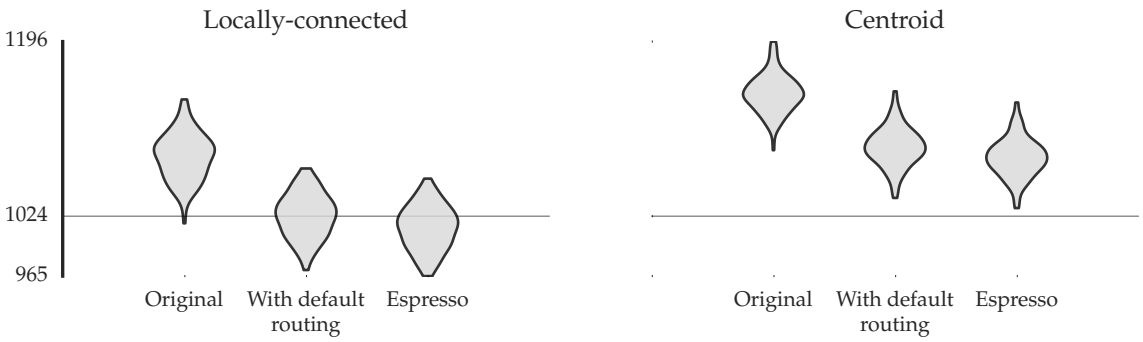


Figure 6.2 – Benchmark routing tables after removing default routes and using Espresso minimisation. Each "violin" represents a vertical histogram of routing table sizes.

"Order-exploiting" minimisation

Unlike IP routing tables, those in SpiNNaker are largely static. As such, minimisation which does not preserve exact equivalence, but instead matches a superset of the routing keys in use, can be used to generate routing tables with far fewer entries. For example, `0001` and `0010` may be combined into a new entry (`00XX`) which matches a superset of the two keys originally matched. In IP routing tables, which are considerably larger than those in SpiNNaker and highly dynamic, this minimisation would be avoided as it (a) requires the minimiser to inspect the entire table during minimisation to avoid breaking the functionality of the table, and (b) slows the updating of the table to modify routes. However, in the case of SpiNNaker – and other systems with static routing tables and tight constraints on entries – this technique allows for significant reductions in table size.

The technique of Liu (2002) was extended to investigate the effect of non-exact minimisation on the benchmark networks. First, each routing table was split into subtables of entries with the same outgoing route. Next the subtables were sorted such that subtables with fewer entries were moved to the top of the routing table and subtables with more toward the bottom. Finally, the minimisation procedure from Espresso was applied to each subtable in turn. However, unlike the technique proposed by Liu (2002), Espresso was allowed to generate entries which matched a superset of keys matched by the original entry provided that they did not 'catch' keys expected to match entries in lower subtables. The minimised table depends on the ordering of the TCAM to ensure that it is functionally equivalent to the original. This use of table ordering to achieve greater compression of entries could be called "order-exploiting" minimisation. Figure 6.3 shows that this new technique for reducing routing table size is sufficient to minimise all the routing tables in the locally-connected benchmark and all but two of the tables in the centroid benchmark to fit within the 1024-entry size required by SpiNNaker.

Default routing cannot generally be used with order-exploiting minimisation, as order-exploiting minimisation could generate entries which match keys which would otherwise be default routed. To avoid this we only minimise tables which contain entries representing every key expected to arrive at the router, including those which would ordinarily be default-routed. As shown in Figure 6.3, benchmark routing tables minimised with order-exploiting minimisation (which do not rely on default routing) are smaller than the original tables with default-routes removed and then minimised as before.

On-chip logic minimisation

Espresso is capable of minimising routing tables of the scale shown in our benchmarks; a mean-time of 6.23 s was spent in Espresso when minimising a table from the locally-connected benchmark on a lightly-loaded Intel Pentium G850. However, as projected SpiNNaker machines are expected to consist of nearly sixty-thousand chips, and the same number of routing tables, the compute time required to minimise all tables is likely to be significant – more than four days if the locally-connected benchmark were scaled to the size of a full SpiNNaker machine. Fortunately, minimisation on this scale is an 'embarrassingly'-parallel problem, and the wall-clock time required to perform this task for SpiNNaker may be significantly reduced by performing the minimisation directly on the chip's own routing tables.

Processing cores in SpiNNaker have access to 64 KiB of Data Tightly-Coupled Memory (DTCM) and 32 KiB of Instruction Tightly-Coupled Memory (ITCM). This places severe limits on the programs which they may execute. While we have seen that Espresso may be used to perform order-exploiting minimisation for use with SpiNNaker, its memory usage and code size preclude its use on embedded systems (Lysecky and Vahid 2003). In contrast, m-Trie minimisation (Ahmad and Mahapatra 2007) has been shown to be capable of effectively compressing IP routing tables while consuming little memory (as low as 16 KiB) but is not suitable for use when order-exploiting minimisation is required (see Figure 6.3) as it will not generate entries which match a superset of the original keys.

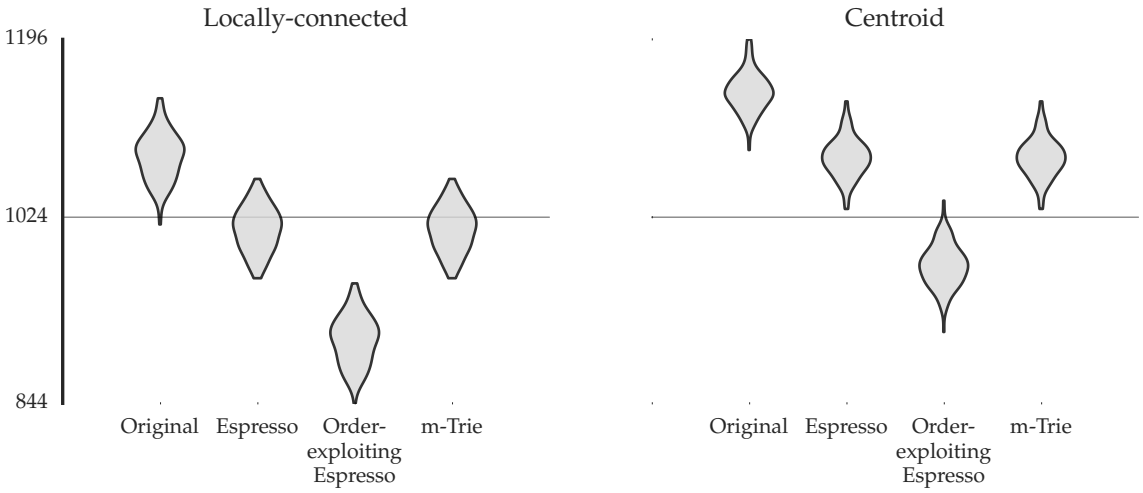


Figure 6.3 – Benchmark performance of order-exploiting Espresso and m-Trie

6.4 Ordered-Covering

This section presents a novel algorithm called "Ordered-Covering" which is capable of performing order-exploiting routing table minimisation within the small memory and code-space available on SpiNNaker. The algorithm proceeds by sequentially *merging* sets of routing table entries with the same output routes while ensuring these merged entries do not *cover* existing entries. A set of routing entries is merged by replacing them with one new entry containing only the bits common to all the original entries and `X`s for all other bits (e.g., `1010` and `0110` would be merged to produce `XX10`). A routing entry covers another when the set of keys matched by one entry intersects with those of another below it in the table.

We now extend the table by annotating each entry in the table with the set of keys that it is expected to match, the *aliases* of the entry. For example, merging the first two entries in the following table would generate a new entry with the original keys listed as aliases:

Key-Mask	Route	Aliases
1011	NE S	1011
0100	NE S	0100
1101	SW 2	1101
1110	SW 2	1110

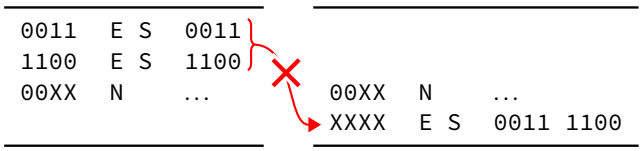
1101	SW 2	1101
1110	SW 2	1110
XXXX	NE S	1011 0100

When merging routing table entries, the following *Ordered-Covering rules* are used to ensure the behaviour of the table remains consistent:

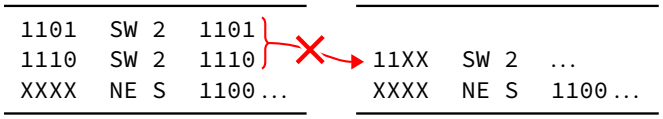
***Up-check* rule:** No entry in the merge may become *covered* by an entry higher up the table. A merge that would be disallowed by this rule is shown in Figure 6.4(a).

***Down-check* rule:** No *aliased entry* below the merge may become *covered*, see Figure 6.4(b).

As an additional heuristic, the table entries are kept sorted in increasing order of *generality*, i.e., the number `X`s contained in their keys and masks. For example, an entry with the key-mask of `00XX` (*generality* of 2) must be placed below any entries with fewer `X`s



(a) A merge which does not obey the *up-check* rule. Before the merge a packet with key `0011` would have been routed to `E S`; after the merge the same packet would be routed to `N` instead. This is because the merge has resulted in the correct entry being moved below an entry which *covers* it.



(b) A merge which does not obey the *down-check* rule. Before the merge a packet with key `1100` would have been routed to `NE S`; after the merge the same packet would be routed to `SW 2` instead. This is because the merge has resulted in the insertion of a new entry which *covers* an existing entry.

Figure 6.4 – Examples of invalid merges as defined by the Ordered-Covering rules

in their key-masks, e.g., below `0000` and `00X1` (generalities of 0 and 1 respectively). Since this is only a partial ordering of the table, we require that new entries of generality g be inserted *above* existing entries of generality g. For example, if `XX00` were already present in the table the new entry `0XX1` must be inserted above it.

Ordered-Covering uses a simple, greedy, algorithm, to minimise a routing table according to these rules:

MINIMIZE(*table*, *target*)

```
1  while table.length > target
2      merge = GET-LARGEST-MERGE(table)
3      if merge.isEmpty
4          break
5      table = APPLY-MERGE(table, merge)
6  return table
```

Where:

- *table* is a routing table that is sorted according to generality heuristics and contains entries for all packets expected to arrive at the router, *including those which could be handled by default routing*

- GET-LARGEST-MERGE is a function which returns a (possibly empty) set of routing table entries which may be merged without breaking the *up-* or *down-check* rules

- APPLY-MERGE merges the set of routing entries and uses the table-ordering heuristics to determine where the new entry should be inserted into the table

The algorithm terminates early if $table.length \leq target$ which can save much time in comparison to fully minimising the table. Commonly, *target* will be set to the number of entries available in the TCAM (e.g., 1024).

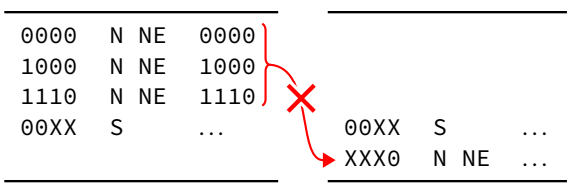
The GET-LARGEST-MERGE function is where the algorithm spends most of its time and a possible implementation is presented in the remainder of this section. This implementation considers merging all groups of entries in the table which share the same set of output ports. In practice, merging these groups will often violate either the up-check or down-check rules. Rather than immediately rejecting these merge candidates, a pair of simple greedy approaches is used to remove entries from merge candidates iteratively until both rules are obeyed. The approach is split into two phases, the first refines merge candidates until they obey the up-check rule and the second does the same for the down-check rule.

Resolving the up-check

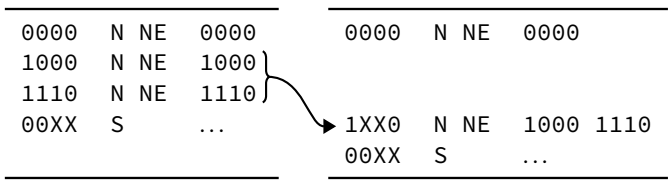
The *up-check* rule ensures that an entry, *e*, cannot – through being merged with other entries – become covered by being moved below another entry which matches a subset of the keys matched by *e*.

For each merge candidate every entry in the merge is inspected to ensure that it does not break the up-check rule. When inspecting an entry, *e*, in the merge candidate we scan through the entries between *e* and the position where the merged entry would be inserted. If any entry is found which would cover *e*, *e* is removed from the merge candidate and we proceed to check the next entry in the candidate merge.

For example, consider the table:



Initially we could consider merging the first three entries. Doing so would mean that `0000` would be combined into the new entry `XXX0`, positioned below `00XX`, causing `0000` to become covered. As the entry matching `0000` is moved below `00XX` by including it in the merge, `0000` must be removed from the merge to avoid covering it. After removing `0000` the, refined, merge is:



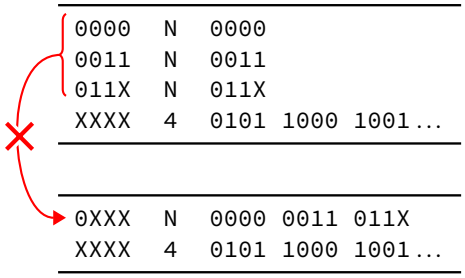
Since the remaining entries in the merge candidate (`1000` and `1110`) are not covered by `00XX` (the only entry between their current positions and the location where the merge is inserted), the refined merge obeys the up-check rule.

Note that, in this case, removing an entry from the merge caused the entry resulting from the merge to be moved upward in the table in accordance with the ordering described above. In cases such as this less work may be required to complete the up-check since the entries which remain in the merge will be nearer their original positions.

Resolving the down-check

The *down-check* rule ensures that the entry resulting from a merge would not match a subset of the packets expected to match entries below the point where the entry resulting from the merge would be located.

For example, consider the table:



Our initial merge candidate, the first three entries, would result in the insertion of the entry `0XXX` immediately above the bottom entry. However, `0XXX` matches `0101`: one of the aliases of `XXXX`. Consequently, merging the first three entries as indicated is not allowed as it violates the down-check rule, changing the behaviour of the routing table.

Once we have identified that the entry resulting from a merge, e_m (e.g., `0XXX`), covers an entry lower in the table, e_c (e.g., `0101`), we attempt to modify the merge to avoid the covering. This may be achieved by converting one of the `X`s in e_m so that it is the opposite of the bit in the same position in e_c. In our example we look to remove entries from the merge such that the new entry resulting from the merge, e'_m, is `0`$\underline{\texttt{0}}$`XX`, `0X`$\underline{\texttt{1}}$`X` or `0XX`$\underline{\texttt{0}}$ – none of which would cover e_c (`0101`).

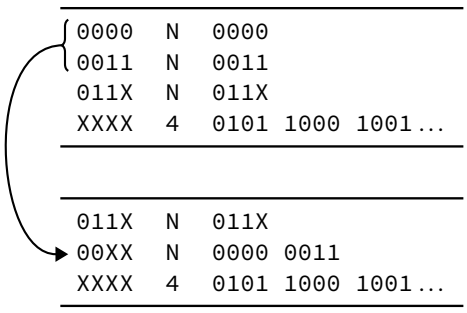
An `X` in the entry resulting from a merge may be converted to another value, v, by removing from the merge any entries which have either `X` or $\overline{v}$ in the given bit position. In our example, to convert `0XX`$\underline{\texttt{X}}$ to `0XX`$\underline{\texttt{0}}$ we remove from the merge any entries with `X` or `1` in the rightmost bit position, in this instance we would remove `001`$\underline{\texttt{1}}$ and `011`$\underline{\texttt{X}}$ from the merge. Alternatively, to convert `0`$\underline{\texttt{X}}$`XX` to `0`$\underline{\texttt{1}}$`1X` we remove from the merge any entries with `X` or `0` in the indicated position, here we would need to remove only `000`$\underline{\texttt{0}}$. Finally, to convert `0`$\underline{\texttt{X}}$`XX` to `0`$\underline{\texttt{0}}$`XX` we would need to remove `0`$\underline{\texttt{1}}$`1X` from the merge. As a result of removing entries we have three new merge candidates:

$$\{\texttt{0000}, \cancel{\texttt{0011}}, \cancel{\texttt{011X}}\} \rightarrow \texttt{000}\underline{\texttt{0}}$$

$$\{\cancel{\texttt{0000}}, \texttt{0011}, \texttt{011X}\} \rightarrow \texttt{0X}\underline{\texttt{1}}\texttt{X}$$

$$\{\texttt{0000}, \texttt{0011}, \cancel{\texttt{011X}}\} \rightarrow \texttt{0}\underline{\texttt{0}}\texttt{XX}$$

We immediately reject the merge candidate with the fewest entries, leaving us with two candidates containing two entries each. As a heuristic, we select the merge candidate which converted the most significant `X` bit, in this case resulting in the merge:



The above process is repeated until either no entries are covered (e_m does not cover any e_c) and a valid merge is produced; or modifying the merge to avoid covering a lower

entry is not possible and the merge must be abandoned. In our example, the new merge candidate results in the insertion of the new entry `00XX` between the two remaining entries. As this new entry does not cover any other entries the algorithm terminates.

Note that removing an entry from the merge to satisfy the down-check might cause the entry resulting from the merge to move upward in the table (in accordance with the table ordering). If this occurs then the down-check must be performed again since additional entries may have become covered. In general the down-check should be repeated until the position of the entry resulting from the merge has stabilised.

In our example there were three bits in e_m, `0XXX`, which could be converted to avoid covering e_c. Precisely which bits may be converted depends upon the nature of the covered entry, for example, if `0XXX` were covering `010X` then only two bits (`0XXX`) may be converted. We prioritize the uncovering of entries for which fewer bits may be converted as a heuristic intended to reduce the number of entries likely to be removed from the merge in the merge candidates considered.

6.5 Results

Compression

Figure 6.5 shows the performance of Ordered-Covering (OC), running on SpiNNaker, when used to minimise the benchmark routing tables. OC reduces all of the tables in the locally-connected benchmark to fit in the router TCAM. While this is not true of the centroid benchmark, OC is able to reduce 66 % of the tables to fewer than 1024 entries. This more than halves the number of tables which would otherwise need to be minimised off-chip. For each minimisation technique the correctness of the minimised routing tables was checked by ensuring that every packet which would be matched by the original table was routed in the same direction by the minimised table.

Memory usage

On-chip routing table minimisation is required to work with small amounts of working memory. In order-exploiting minimisation this small amount of memory must be used to store the whole routing table – e.g., 24 KiB for a 2048-entry table – and any additional structures. Consequently, those additional structures must be memory efficient. After the

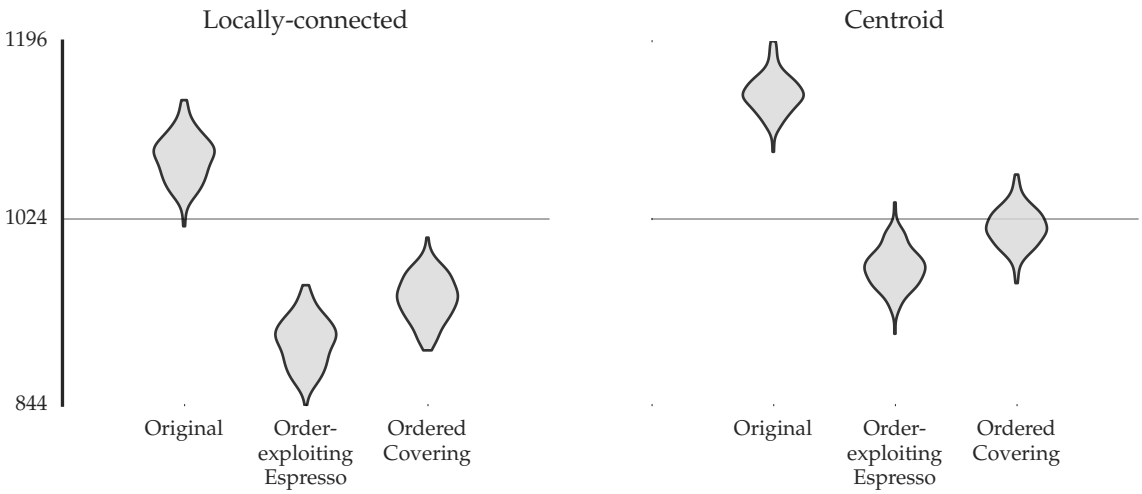


Figure 6.5 – Performance of Ordered-Covering when used to minimise benchmark tables. Ordered-Covering minimises all of the tables in the locally-connected benchmark, and 66 % of the tables in the centroid benchmark, to ≤ 1024 entries.

routing table, the aliases table (which annotates routing table entries with the keys they are expected to match) is the most expensive structure in Ordered-Covering. Other structures, such as sets of entries being considered as merge candidates can be implemented using bit-vectors and consume little memory.

Each merge of n entries performed by Ordered-Covering reduces the size of the routing table by $n - 1$ entries (reducing the memory required by $3(n - 1)$ words) and adds a new entry into the aliases table. In this implementation of the aliases table, which uses AA-trees (Andersson 1993) and linked-lists of fixed-size blocks, a merge of n entries will increase the memory required by the aliases table by $2(n + 1) + 4$ words. Consequently, the total memory cost of a merge of n entries is $2(n + 1) + 4 - 3(n - 1) = 9 - n$ words. If the space saved by minimising the routing table were reclaimed then any merge of ten or more entries would reduce the memory required by the routing and aliases tables.

The demonstrated implementation of Ordered-Covering does not reclaim the memory saved by reducing the size of a routing table, thus the routing table remains a constant size while the aliases table grows with every merge. The peak heap usage of this minimiser when running on SpiNNaker was 18.4 KiB for the locally-connected benchmark (of which 13.3 KiB was used to represent the routing table) and 18.8 KiB for the centroid model (14.0 KiB for the routing table). Ordered-Covering is able to achieve good levels of table compression using little more memory than that required to store the routing table.

Execution time

Table 6.1 shows the time required to load and minimise the benchmark routing tables using Ordered-Covering on SpiNNaker with routing tables minimised in parallel across the machine. Exploiting the parallelism of SpiNNaker ensures that while increasing the size of the target machine would increase the time taken to load the tables it would not increase the time taken to minimise them (for some patterns of network connectivity). In contrast, Table 6.2 shows the time taken to minimise the same benchmarks serially on an Intel Pentium G850 using order-exploiting Espresso.

Model	Load time / s	Exec. time / s	
		Sufficient	Fully
Locally-connected	3.8	13.9	25.6
Centroid	3.6		25.6

Table 6.1 – Time to load and minimise 144 benchmark tables using Ordered-Covering on SpiNNaker (N.B.: Not all tables were sufficiently minimised in the centroid model)

Model	Cumulative time / s	Mean time per table / s
Locally-connected	897.2	6.2
Centroid	1146.8	8.0

Table 6.2 – Time to minimise 144 benchmark tables using order-exploiting Espresso on an Intel Pentium G850

Since Ordered-Covering was able to minimise all tables in the locally-connected benchmark to fit within TCAM no tables would need to be minimised off-chip. Consequently, on-chip OC represents a 64.5× speed-up compared to using order-exploiting Espresso.

Ordered-Covering was only able to minimise 66 % of routing tables in the centroid benchmark to fit in TCAM. All but two of the remainder of these tables could then be minimised using order-exploiting Espresso, and the total time taken to minimise tables using this joint approach would be around 413 s. In this instance combined use of OC and Espresso represents a 2.8× speed-up compared to just using order-exploiting Espresso.

As routing tables may be minimised in parallel, the time required by order-exploiting Espresso could be reduced by using more processing cores. However, significantly reducing the time taken to minimise routing tables of the largest SpiNNaker machine (57 600 tables) will prove challenging without making use of the machine itself.

6.6 Summary

This chapter has demonstrated that routing table minimisation for SpiNNaker is a challenging problem. In particular, it was shown that neither 'default routing' nor existing minimisation techniques were able to reduce tables from two benchmark models to fit within the limited number of TCAM entries in a router. This could be overcome by extending the minimisation technique of Liu (2002) to use Espresso to generate functionally equivalent, compressed, routing tables which matched a superset of the packets matched by the original tables and fit within the available number of entries. However, due to the limited code-space and memory available to each SpiNNaker core, this technique was not suitable for direct implementation on-chip: a desirable feature as routing table minimisation is 'embarassingly'-parallel and the largest SpiNNaker system will contain nearly sixty-thousand routing tables.

To overcome these issues "Ordered-Covering" was introduced, a novel algorithm for routing table minimisation capable of running within the small amount of memory available to a SpiNNaker processor. Using Ordered-Covering, in parallel on SpiNNaker, it was possible to minimise all the tables in one of the benchmarks in 17.7 s – a speed-up of 64.5× compared to off-chip minimisation using Espresso. In the other benchmark, combined use of on- and off-chip minimisation using Ordered-Covering and Espresso was able to achieve a speed-up of 2.8× of the time taken by off-chip minimisation alone.

The high degrees of fan-in and fan-out characteristic of the circular convolution network described in Section 5.3 were only partially reduced through the introduction of interposers. Representing the fan-in or fan-out of many packets with differing keys requires large routing tables and since these tables are not always sufficiently small to store in hardware they must be compressed. This chapter introduced techniques which improved the compression of these tables, making simulation of networks like circular convolution possible. Furthermore, performing the minimisation on SpiNNaker reduces the energy consumed and the time required – further reducing the cost of simulating neural models such as Spaun.

Chapter 7

Conclusion – Spaun and SpiNNaker

Simulating models of the brain is of increasing importance to neuroscience and the cognitive sciences (Markram et al. 2011; The White House 2013). The scale and complexity of models which can be simulated is limited by the time, computational resource and energy available. In addition, although it is desirable that neural simulations be interfaced with the growing range of biologically-inspired sensors and actuators to enable more investigation of the role of environment in cognition, this requires simulations which run in biological real time and has only been possible for small models.

Spaun (Eliasmith, Stewart, et al. 2012) is, arguably, the current leading *functional* brain model. Built using the Neural Engineering Framework (NEF) (Eliasmith and Anderson 2004) and the Semantic Pointer Architecture (Eliasmith 2013), Spaun illustrates how a number of cognitive and non-cognitive tasks can be performed by a neurally-inspired implementation of a cognitive architecture. Unfortunately, 1 s of its simulation required 2.5 h on a large compute cluster (Stewart and Eliasmith 2014). While it has been informally reported that this time has been reduced by around an order of magnitude, simulation of the model remains significantly slower than biology. Consequently, there is limited scope to construct and simulate models larger or more complex than Spaun; it is expensive to run sufficient experiments to increase confidence in the claims made for Spaun, and, finally, Spaun runs too slowly to interact with the real world.

SpiNNaker is a massively parallel, low power, computer specifically designed for the simulation of spiking neural networks. Unfortunately, neural models built using the NEF apparently mapped poorly to the SpiNNaker architecture, limiting its capability to accelerate large scale models such as Spaun.

The purpose of this thesis was to investigate whether the Spaun model of cognition could be simulated in real time on the SpiNNaker architecture. Three techniques which facilitate this have been presented:

1. Characteristics of networks built using the Neural Engineering Framework (NEF) act to stress the SpiNNaker architecture. Specifically, storing the large, dense synaptic weight matrices that result from the NEF consumes much more memory than is available to a SpiNNaker core and high neural firing rates result in heavy network traffic and significant compute loads. This resulted in poor use of the architecture. In Chapter 4 it was demonstrated how these costs could be reduced by exploiting the fact that the synaptic weight matrices can be *factored*. Not only does this reduce the data stored by each core but it allows the transformation of the high dimensional spikes into a lower dimensional space. This transformation reduces the network and compute resources required to simulate these neural networks.

2. While the technique presented in Chapter 4 significantly reduces the compute resource consumed by a core component of the Spaun model (Table 7.1) the network traffic required is still greater than could be supported for real time simulation. In Chapter 5 it was demonstrated that further significant reductions in traffic and compute load could be achieved by using an 'interposer', rather than the transmitting core, to transform the lower dimensional values transmitted between processors. These reductions were sufficient that real time simulation of this core component is possible, albeit at the cost of around two hundred more processors. This balance between compute and communication resource will be revisited later.

3. Chapters 4 and 5 dealt with the computation, memory and communication loads posed by the Neural Engineering Framework and the Semantic Pointer Architecture to SpiNNaker. However, communicating simulation state across the machine requires that the routing information be represented in tables of limited size and a core component of Spaun results in routing tables which are too large. Chapter 6 proposed ways that the size of these tables could be reduced using logic minimisation. A new algorithm was introduced which exploits the massive parallelism of SpiNNaker to reduce the time required to minimise routing tables. These techniques are also applicable to reduce the size of the tables which route values to synaptic filters (Section 4.2).

Together, these techniques address the memory, compute, network and routing costs resulting from distributing spikes across a SpiNNaker machine, and facilitate real time simulation of Spaun. This is an estimated four times faster than if SpiNNaker had been used as intended (extrapolating from Table 7.1) and 9000 times faster than the original experiments (Stewart and Eliasmith 2014). From figures presented in Table 7.1 it is clear that use of SpiNNaker without the techniques presented in this thesis would still allow for a significant acceleration of the Spaun model. This is not surprising given the massive parallelism and cheap communication of the SpiNNaker architecture. However, such a simulation would require nearly 50 % of the largest possible SpiNNaker machine – significantly reducing the capacity to increase the scale and complexity of Spaun. Moreover, the simulation could run at only a quarter of biological real time – limiting the extent to which Spaun could interact with the real world using neuromorphic sensors and actuators. The techniques presented in this thesis overcome these obstacles to deliver simulation in biological real time using far fewer resources.

Method	Cores required	Percentage of real time / %
SpiNNaker as intended (estimated)	66 720	25
After Chapter 4[†]	2440	20
After Chapter 5[†]	2620	100

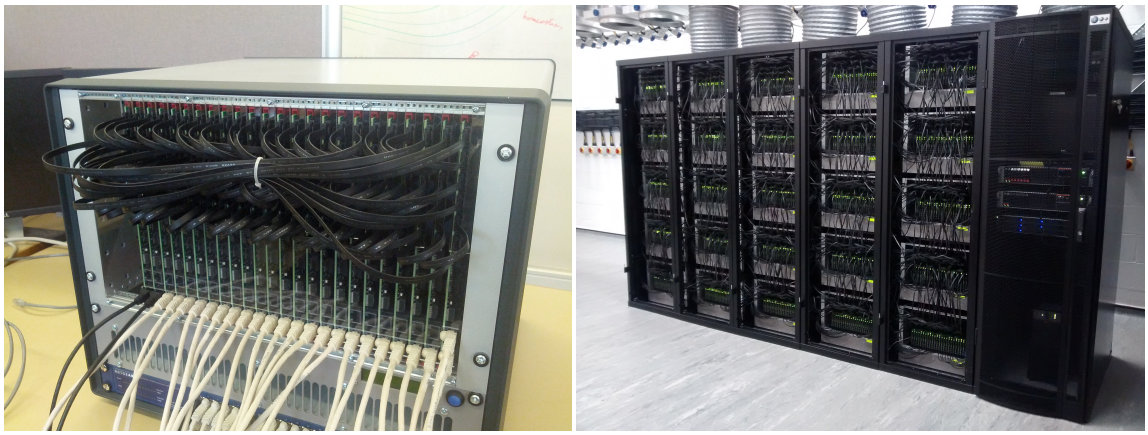
[†]Only possible as a result of the routing table minimisation of Chapter 6.

Table 7.1 – Comparison of simulation methods for a core Spaun component (circular convolution). Figures are for the fastest possible simulation or biological real time. Reproduced from Table 5.6.

Since the techniques presented in this thesis require only one twentieth of the SpiNNaker resources that would otherwise be needed to simulate a core component of Spaun, it is estimated that Spaun can be simulated on the 'desktop-sized' SpiNNaker machine shown in Figure 7.1(a) rather than the 'room-sized' machine of Figure 7.1(b).

7.1 SpiNNaker

Much of this thesis has been concerned with mapping a challenging problem to the SpiNNaker platform. SpiNNaker was intended specifically for the simulation of spiking neural networks and this is reflected in many of the design choices. For example, the SpiNNaker network is optimised for the transmission of short (usually 40 bit, in this thesis



(a) **(b)**

Figure 7.1 – SpiNNaker machines. A 1152-chip, twenty thousand core, SpiNNaker 'frame' is shown in (a). In (b) 25 of these frames have been combined to construct a half-million core SpiNNaker machine which is being expanded to over a million processors.

72 bit), multicast, packets. Since avoiding deadlock by re-routing stuck packets is hard, packets are instead discarded if the network becomes blocked – consequently, packet delivery is not guaranteed. These choices echo the nature of the brain, in which the spikes of inherently noisy and unreliable neurons are distributed to many connected peers. Likewise, the lack of hardware support for floating point calculation was justified on the grounds that, as brains must be error correcting and noise tolerant, fixed point representations of neuron and synapse state would be sufficient (Furber and Temple 2007).

Although the Neural Engineering Framework results in networks which have been shown to map poorly to the architecture *as it was intended to be used* it has been possible to recast the problem in a way which works well. In particular, Chapter 4 exploited the *payloads* of SpiNNaker packets to deliver *decodings* of neural activity and Chapter 5 effectively used the *multicast* capability of the SpiNNaker network to significantly reduce network traffic. However, the techniques demonstrated in these chapters are pragmatic compromises. For example, the loss of a packet containing decoded neural activity is worse than the loss of a single spike packet since it necessarily carries more information. Moreover, factoring the synaptic weight matrices and communicating with decodings of neural activity requires that more fixed point multiplications are performed than would be required if the full synaptic weight matrices were used. The interposer technique exacerbates this problem by introducing another stage of multiplication. While issues related to fixed point representations could be avoided by changes to future SpiNNaker

hardware it is unclear that a trivial fix for the problem of dropped packets exists.

The use of the payloads of packets to transmit decodings of neural activity has been shown to reduce network traffic, however, it also increases the effect of any dropped packet since payloads contain the contributions of *many* spikes. Any loss is mitigated, to some degree, by the presence of synaptic filters which will 'smooth out' a few missing packets and, in the Semantic Pointer Architecture, the high dimensional vectors used to represent symbols ensure that there is some redundancy. Thus, while infrequent packet loss should not be expected to change the *behaviour* of the neural model, any repeated or widespread loss of packets will significantly reduce the accuracy of a simulation. Consequently, it has been important to ensure that as few packets as possible are transmitted (Chapters 4 and 5), that packet transmission is not bursty (Chapter 4) and that the bandwidth of SpiNNaker links is not exceeded (Chapter 5). Although software support for reinserting dropped packets is under development, this will only improve the reliability of bursty traffic and cannot overcome the fundamental bandwidth limits of physical links. Therefore, careful application design (e.g., this thesis; Knight 2016) should be paired with effective place-and-route (Heathcote 2016).

The lack of floating point hardware in SpiNNaker forces, for performance reasons, the use of fixed point arithmetic with the resultant loss of precision. This use of fixed point arithmetic does appear to reduce, slightly, the accuracy of simulations presented in this thesis (Chapter 4) and other larger models which have not been presented. Despite this, it was shown in Chapter 5 that the likelihood of confusing two high-dimensional vectors remained vanishingly small and thus the higher-level *behaviour* of Spaun should be expected to be unaffected by the limited precision arithmetic available on SpiNNaker. Unfortunately, the magnitude of the decoders of an ensemble is inversely proportional to the size of the ensemble (the more neurons an ensemble includes the smaller to contribution of each neuron to the decoding) hence, as the size of ensembles grows the error due to fixed point rounding will increase. On current generation SpiNNaker hardware this could be mitigated by rescaling decoders to increase their precision and downscaling the resulted decoded values prior to transmission. However, as there remains a trade off between precision and range in any fixed point representation the author believes that floating point support would be of significant benefit in any future version of the SpiNNaker hardware.

Finally, the short packets used on SpiNNaker result in inefficient transmission of vec-

tors over the network. Each element of the vector necessitates the transmission of an 8 bit header and a 32 bit key in addition to the 32 bit payload. Consequently, for a 16-D vector, 144 B are required to transmit a 64 B value. Allowing larger payloads would, at some cost in network complexity, significantly reduce the bandwidth required to transmit these high dimensional values.

Despite the potential pitfalls outlined above, the author contends that SpiNNaker is an excellent platform for the simulation of neural models like Spaun.

7.2 Spaun

The Neural Engineering Framework (NEF) and Semantic Pointer Architecture (SPA) are two hypotheses about how the brain might represent and manipulate vectors and symbols. Their union, in Spaun (Eliasmith, Stewart, et al. 2012), has previously demonstrated that cognitive architectures can be implemented a neurally-inspired medium, tying together what were previously disparate threads of the cognitive sciences.

Unfortunately, the two characteristics that result in the initially poor mapping of NEF to the SpiNNaker architecture, namely high firing rates and dense synaptic weight matrices, also call into question the biological plausibility of the framework. A full analysis of this claim is beyond the scope of this thesis, however, the ability to perform large scale simulations of complex NEF models in biological real time which has been facilitated by this thesis makes it considerably easier to generate data to assess claims both for and against the NEF, SPA and Spaun.

7.3 Future work

A few small engineering problems remain before Spaun can be simulated on SpiNNaker using the Nengo (Bekolay et al. 2014) backend[1] that was developed for this thesis but it is expected that simulation of the full model will occur within the year. Further benchmarking of the performance of the techniques presented in this thesis will also be required.

Longer term, the work performed for this thesis identifies areas in which improvements can be made to subsequent versions of the SpiNNaker hardware. These observations are being included in the design of the next SpiNNaker chip.

[1] https://github.com/project-rig/nengo_spinnaker

The author intends to join the research group which developed Spaun, where efforts are underway to build more complex and more comprehensive functional brain models. These models will exploit the large scale, real time simulation made possible by this thesis.

7.4 Summary

This thesis investigated, and has presented three techniques which facilitate, the real time simulation of Spaun on SpiNNaker. This represents a major milestone for both projects since it demonstrates the capability of SpiNNaker to support neural models of truly significant scale and provides a way for more complex and detailed NEF models to be simulated.

References

Abbott, L. F. (1999). "Lapicque's introduction of the integrate-and-fire model neuron (1907)". In: *Brain Research Bulletin* 50 (5-6), pp. 303–304. ISSN: 0361-9230. DOI: 10.1016/S0361-9230(99)00161-6.

Ahmad, S. and R.N. Mahapatra (2007). "An Efficient Approach to On-Chip Logic Minimization". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 15 (9), pp. 1040–1050. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2007.902202.

Amdahl, Gene M. (1967). "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, pp. 483–485. DOI: 10.1145/1465482.1465560.

Ananthanarayanan, Rajagopal, Steven K Esser, Horst D Simon, and Dharmendra S Modha (2009). "The cat is out of the bag: cortical simulations with 10^9 neurons, 10^{13} synapses". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, pp. 1–12.

Anderson, John R et al. (2004). "An integrated theory of the mind." In: *Psychological review* 111 (4), pp. 1036–1060. ISSN: 0033-295X. DOI: 10.1037/0033-295X.111.4.1036.

Andersson, Arne (1993). "Balanced search trees made simple". In: *Algorithms and Data Structures*. Ed. by Frank Dehne, Jörg-Rüdige Sack, Nicola Santoro, and Sue Whitesides. Springer, pp. 60–71. DOI: 10.1007/3-540-57155-8.

Bassett, Danielle S. et al. (2010). "Efficient Physical Embedding of Topologically Complex Information Processing Networks in Brains and Computer Circuits". In: *PLOS Computational Biology* 6 (4), pp. 1–14. DOI: 10.1371/journal.pcbi.1000748.

Bednar, James A (2009). "Topographica: building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components". In: *Frontiers in Neuroinformatics* 3. ISSN: 1662-5196. DOI: 10.3389/neuro.11.008.2009.

Bekolay, Trevor et al. (2014). "Nengo: A Python tool for building large-scale functional brain models". In: *Frontiers in Neuroinformatics* 7 (48). ISSN: 1662-5196. DOI: 10.3389/fninf.2013.00048.

Berzish, Murphy, Chris Eliasmith, and Bryan Tripp (2016). "Real-Time FPGA Simulation of Surrogate Models of Large Spiking Networks". In: *Artificial Neural Networks and Machine Learning – ICANN 2016: 25th International Conference on Artificial Neural Networks, Barcelona, Spain, September 6-9, 2016, Proceedings, Part I*. Ed. by Alessandro E.P. Villa, Paolo Masulli, and Antonio Javier Pons Rivero. Cham: Springer International Publishing, pp. 349–356. ISBN: 978-3-319-44778-0. DOI: 10.1007/978-3-319-44778-0_41.

Boahen, Kwabena et al. (2000). "Point-to-point connectivity between neuromorphic chips using address events". In: *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on* 47 (5), pp. 416–434.

Braitenberg, Valentino (1986). *Vehicles: Experiments in synthetic psychology*. Cambridge, MA: MIT. ISBN: 978-0-26252-112-3.

Brayton, Robert K (1984). *Logic minimization algorithms for VLSI synthesis*. Vol. 2. The Springer International Series in Engineering and Computer Science. Springer Science & Business Media. ISBN: 978-0-89838-164-1. DOI: 10.1007/978-1-4613-2821-6.

Brette, Romain et al. (2007). "Simulation of networks of spiking neurons: a review of tools and strategies". In: *Journal of computational neuroscience* 23 (3), pp. 349–398.

Brooks, Rodney A. (1990). "Elephants don't play chess". In: *Robotics and Autonomous Systems* 6 (1), pp. 3–15. ISSN: 0921-8890. DOI: 10.1016/S0921-8890(05)80025-9.

Carnevale, Nicholas T and Michael L Hines (2006). *The NEURON book*. Cambridge, England: Cambridge University Press. ISBN: 978-0-51154-161-2.

Cassidy, Andrew, Andreas G Andreou, and Julius Georgiou (2011). "Design of a one million neuron single FPGA neuromorphic system for real-time multimodal scene analysis". In: *Information Sciences and Systems (CISS), 2011 45th Annual Conference on*. IEEE, pp. 1–6.

Chan, V., Shih-Chii Liu, and A. van Schaik (2007). "AER EAR: A Matched Silicon Cochlea Pair With Address Event Representation Interface". In: *Circuits and Systems I: Regular Papers, IEEE Transactions on* 54 (1), pp. 48–59. ISSN: 1549-8328. DOI: 10.1109/TCSI.2006.887979.

Choudhary, Swadesh et al. (2012). "Silicon neurons that compute". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7552 LNCS. PART 1, pp. 121–128.

Cormen, Thomas H (2009). *Introduction to algorithms*. 3rd ed. Cambridge, MA: MIT press. ISBN: 978-0-26253-305-8.

Cox, Charles E and W Ekkehard Blanz (1992). "GANGLION-a fast field-programmable gate array implementation of a connectionist classifier". In: *IEEE Journal of Solid-State Circuits* 27 (3), pp. 288–299.

Crawford, Eric, Matthew Gingerich, and Chris Eliasmith (2013). "Biologically Plausible, Human-scale Knowledge Representation". In: *35th Annual Conference of the Cognitive Science Society*, pp. 412–417.

Davies, Sergio, Javier Navaridas, Francesco Galluppi, and Steve Furber (2012). "Population-based routing in the SpiNNaker neuromorphic architecture". In: *Neural Networks (IJCNN), The 2012 International Joint Conference on*. IEEE, pp. 1–8.

Davison, Andrew P et al. (2008). "PyNN: a common interface for neuronal network simulators". In: *Frontiers in Neuroinformatics* 2. ISSN: 1662-5196. DOI: 10.3389/neuro.11.011.2008.

Dayan, Peter and Laurence F Abbott (2001). *Theoretical neuroscience*. Cambridge, MA: MIT press. ISBN: 978-0-26254-185-5.

Diamond, Alan, Thomas Nowotny, and Michael Schmuker (2015). "Comparing Neuromorphic Solutions in Action: Implementing a Bio-Inspired Solution to a Benchmark Classification Task on Three Parallel-Computing Platforms". In: *Frontiers in neuroscience* 9.

Dongarra, Jack et al. (2016). "Parallel Programming Models for Dense Linear Algebra on Heterogeneous Systems". In: *Supercomputing frontiers and innovations* 2 (4), pp. 67–86.

Dosher, Barbara Anne (1999). "Item interference and time delays in working memory: Immediate serial recall". In: *International Journal of Psychology* 34 (5-6), pp. 276–284.

Eliasmith, Chris (2005). "A unified approach to building and controlling spiking attractor networks." In: *Neural computation* 17 (6), pp. 1276–1314. ISSN: 0899-7667. DOI: 10.1162/0899766053630332.

— (2013). *How to build a brain: A neural architecture for biological cognition*. Oxford, UK: Oxford University Press. ISBN: 978-0-19979-469-0.

Eliasmith, Chris and Charles H Anderson (2004). *Neural Engineering*. Cambridge, MA: MIT Press. ISBN: 978-026255-060-4.

Eliasmith, Chris, Jan Gosmann, and Xuan Choo (2016). "BioSpaun: A large-scale behaving brain model with complex neurons". In: *arXiv preprint arXiv:1602.05220*.

Eliasmith, Chris, Terrence C Stewart, et al. (2012). "A large-scale model of the functioning brain". In: *Science* 338 (6111), pp. 1202–1205.

Fidjeland, Andreas K and Murray P Shanahan (2010). "Accelerated simulation of spiking neural networks using GPUs". In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*. IEEE, pp. 1–8.

Fodor, Jerry A and Zenon W Pylyshyn (1988). "Connectionism and cognitive architecture: A critical analysis. Special Issue: Connectionism and symbol systems". In: *Cognition* 28 (1-2), pp. 3–71.

Furber, S., F. Galluppi, S. Temple, and L.A. Plana (2014). "The SpiNNaker Project". In: *Proceedings of the IEEE* 102 (5), pp. 652–665. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2304638.

Furber, S. and S. Temple (2007). "Neural systems engineering". In: *Journal of The Royal Society Interface* 4 (13), pp. 193–206. ISSN: 1742-5689. DOI: 10.1098/rsif.2006.0177.

Galluppi, F., S. Davies, S. Furber, T. Stewart, and C. Eliasmith (2012). "Real time on-chip implementation of dynamical systems with spiking neurons". In: *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pp. 1–8. DOI: 10.1109/IJCNN.2012.6252706.

Gelder, Tim van (1998). "The dynamical hypothesis in cognitive science." In: *The Behavioral and brain sciences* 21 (5), 615–628, discussion 629–665. ISSN: 0140-525X. DOI: 10.1017/S0140525X98001733.

Gewaltig, Marc-Oliver and Markus Diesmann (2007). "NEST (NEural Simulation Tool)". In: *Scholarpedia* 2 (4), p. 1430.

Goodman, Dan F M and Romain Brette (2009). "The Brian simulator". In: *Frontiers in Neuroscience* 3 (26). ISSN: 1662-453X. DOI: 10.3389/neuro.01.026.2009.

Gosmann, Jan (2015). *Precise multiplications with the NEF*. URL: https://github.com/ctn-archive/technical-reports/blob/master/Precise-multiplications-with-the-NEF.ipynb.

Gosmann, Jan and Chris Eliasmith (2016). "Optimizing Semantic Pointer Representations for Symbol-Like Processing in Spiking Neural Networks". In: *PLOS ONE* 11 (2), pp. 1–18. DOI: 10.1371/journal.pone.0149928.

Gurney, Kevin, Tony J Prescott, and Peter Redgrave (2001). "A computational model of action selection in the basal ganglia. I. A new functional anatomy". In: *Biological cybernetics* 84 (6), pp. 401–410.

Heathcote, Jonathan (2016). "Building and operating large-scale SpiNNaker machines". PhD thesis. Manchester, UK: School of Computer Science, University of Manchester.

Hellwig, Bernhard (2000). "A quantitative analysis of the local connectivity between pyramidal neurons in layers 2/3 of the rat visual cortex". In: *Biological Cybernetics* 82 (2), pp. 111–121. ISSN: 1432-0770. DOI: 10.1007/PL00007964.

Herculano-Houzel, Suzana (2009). "The human brain in numbers: a linearly scaled-up primate brain." English. In: *Frontiers in human neuroscience* 3, p. 31. ISSN: 1662-5161. DOI: 10.3389/neuro.09.031.2009.

Hinton, Geoffrey E and Ruslan R Salakhutdinov (2006). "Reducing the dimensionality of data with neural networks". In: *Science* 313 (5786), pp. 504–507.

Hodgkin, Alan L and Andrew F Huxley (1952). "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *The Journal of physiology* 117 (4), p. 500.

Hopfield, J J (1982). "Neural networks and physical systems with emergent collective computational abilities". In: *Proceedings of the National Academy of Sciences* 79 (8), pp. 2554–2558. eprint: http://www.pnas.org/content/79/8/2554.full.pdf.

Hopkins, Michael and Steve Furber (2015). "Accuracy and efficiency in fixed-point neural ODE solvers". In: *Neural computation* 27, pp. 2148–2182.

Indiveri, Giacomo et al. (2011). "Neuromorphic silicon neuron circuits". In: *Frontiers in neuroscience* 5, p. 73.

Izhikevich, Eugene M (2004). "Which model to use for cortical spiking neurons?" In: *IEEE transactions on neural networks* 15 (5), pp. 1063–70. ISSN: 1045-9227. DOI: 10.1109/TNN.2004.832719.

Izhikevich, Eugene M and Gerald M Edelman (2008). "Large-scale model of mammalian thalamocortical systems." In: *Proceedings of the National Academy of Sciences of the United States of America* 105 (9), pp. 3593–3598.

Kandel, Eric R. (1976). *Cellular Basis of Behavior: An Introduction to Behavioral Neurobiology*. W H Freeman Limited, p. 727. ISBN: 0716705222.

Kaplan, Frederic (2008). "Neurorobotics: an experimental science of embodiment". In: *Frontiers in neuroscience* 2 (1), p. 22.

Keckler, Stephen W, William J Dally, Brucek Khailany, Michael Garland, and David Glasco (2011). "GPUs and the future of parallel computing". In: *IEEE Micro* 31 (5), pp. 7–17.

Kindratenko, Volodymyr V et al. (2009). "GPU clusters for high-performance computing". In: *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, pp. 1–8.

Knight, James (2016). "Plasticity in large-scale neuromorphic models of the neocortex". PhD thesis. Manchester, UK: School of Computer Science, University of Manchester.

Knight, James C. and Steve B. Furber (2016). "Synapse-Centric Mapping of Cortical Models to the SpiNNaker Neuromorphic Architecture". In: *Frontiers in Neuroscience* 10, p. 420. ISSN: 1662-453X. DOI: 10.3389/fnins.2016.00420.

Knight, James C, Philip J Tully, Bernhard A Kaplan, Anders Lansner, and Steve B Furber (2016). "Large-scale simulations of plastic neural networks on neuromorphic hardware". In: *Frontiers in neuroanatomy* 10.

Knight, James, Aaron R. Voelker, Andrew Mundy, and Chris Eliasmith (2016). "Efficient SpiNNaker simulation of a heteroassociative memory using the Neural Engineering Framework". In: *Neural Networks (IJCNN), 2016 International Joint Conference on*.

Kung, Hsiang Tsung and Charles E Leiserson (1980). "Algorithms for VLSI processor arrays". In: *Introduction to VLSI systems*, pp. 271–292.

Laird, John E, Allen Newell, and Paul S Rosenbloom (1987). "Soar: An architecture for general intelligence". In: *Artificial intelligence* 33 (1), pp. 1–64.

Lapicque, Louis (1907). "Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation". In: *J. Physiol. Pathol. Gen* 9, pp. 620–635.

Lichtsteiner, P., C. Posch, and T. Delbruck (2008). "A 128 $\times$ 128 120 dB 15 µs Latency Asynchronous Temporal Contrast Vision Sensor". In: *Solid-State Circuits, IEEE Journal of* 43 (2), pp. 566–576. ISSN: 0018-9200. DOI: 10.1109/JSSC.2007.914337.

Liu, Huan (2002). "Routing table compaction in ternary CAM". In: *Micro, IEEE* 22 (1), pp. 58–64. ISSN: 0272-1732. DOI: 10.1109/40.988690.

Lysecky, R. and F. Vahid (2003). "On-chip logic minimization". In: *Design Automation Conference, 2003. Proceedings*, pp. 334–337. DOI: 10.1109/DAC.2003.1219019.

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

Markram, Henry (2006). "The blue brain project". In: *Nature Reviews Neuroscience* 7 (2), pp. 153–160.

Markram, Henry et al. (2011). "Introducing the Human Brain Project". In: *Procedia Computer Science*. Vol. 7, pp. 39–42.

McCulloch, W S and W Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biophysics* 5 (4), pp. 115–133. ISSN: 00928240. DOI: 10.1007/BF02478259.

Mead, Carver (1989). *Analog VLSI and neural systems*. Addison-Wesley, p. 371. ISBN: 0201059924.

Meier, Karlheinz (2015). "A mixed-signal universal neuromorphic computing system". In: *2015 IEEE International Electron Devices Meeting (IEDM)*. IEEE, pp. 4–6.

Merolla, Paul A. et al. (2014). "A million spiking-neuron integrated circuit with a scalable communication network and interface". In: *Science* 345 (6197), pp. 668–673. DOI: 10.1126/science.1254642.

Moore, Simon W, Paul J Fox, Steven JT Marsh, A Theodore Markettos, and Alan Mujumdar (2012). "Bluehive-a field-programable custom computing machine for extreme-scale real-time neural network simulation". In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, pp. 133–140.

Morrison, Abigail, Carsten Mehring, Theo Geisel, AD Aertsen, and Markus Diesmann (2005). "Advancing the boundaries of high-connectivity network simulation with distributed computing". In: *Neural computation* 17 (8), pp. 1776–1801.

Mundy, Andrew, Jonathan Heathcote, and Jim D. Garside (2016). "On-Chip Order-Exploiting Routing Table Minimization for a Multicast Supercomputer Network". In: *High Performance Switching and Routing (HPSR), 17th International Conference on*.

Mundy, Andrew, James Knight, Terrence C. Stewart, and Steve Furber (2015). "An efficient SpiNNaker implementation of the Neural Engineering Framework". In: *Neural Networks (IJCNN), 2015 International Joint Conference on*. DOI: 10.1109/IJCNN.2015.7280390.

Nageswaran, Jayram Moorkanikara, Nikil Dutt, Jeffrey L Krichmar, Alex Nicolau, and Alexander V Veidenbaum (2009). "A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors". In: *Neural networks* 22 (5), pp. 791–800.

Navaridas, Javier, Mikel Luján, Luis A. Plana, Steve Temple, and Steve B. Furber (2015). "SpiNNaker: Enhanced multicast routing". In: *Parallel Computing* 45. Computing Frontiers 2014: Best Papers, pp. 49–66. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2015.01.002.

Pallipuram, Vivek K, Mohammad Bhuiyan, and Melissa C Smith (2012). "A comparative study of GPU programming models and architectures using neural networks". In: *The Journal of Supercomputing* 61 (3), pp. 673–718.

Plate, Tony A. (1995). "Holographic reduced representations." In: *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 6 (3), pp. 623–41. ISSN: 1045-9227. DOI: 10.1109/72.377968.

Popper, K.R. (1979). *Objective Knowledge: An Evolutionary Approach*. Clarendon Press. ISBN: 9780198750246.

Potjans, Tobias C and Markus Diesmann (2012). "The Cell-Type Specific Cortical Microcircuit: Relating Structure and Activity in a Full-Scale Spiking Network Model." In: *Cerebral cortex (New York, N.Y. : 1991)*. ISSN: 1460-2199. DOI: 10.1093/cercor/bhs358.

Rasmussen, Daniel and Chris Eliasmith (2014). "A spiking neural model applied to the study of human performance and cognitive decline on Raven's Advanced Progressive Matrices". In: *Intelligence* 42 (1), pp. 53–82.

Redgrave, Peter, Tony J Prescott, and Kevin Gurney (1999). "The basal ganglia: a vertebrate solution to the selection problem?" In: *Neuroscience* 89 (4), pp. 1009–1023.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain". In: *Psychological review* 65 (6), p. 386.

Sandamirskaya, Yulia (2013). "Dynamic neural fields as a step toward cognitive neuromorphic architectures." English. In: *Frontiers in neuroscience* 7, p. 276. ISSN: 1662-4548. DOI: 10.3389/fnins.2013.00276.

Schemmel, Johannes et al. (2010). "A wafer-scale neuromorphic hardware system for large-scale neural modeling". In: *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. IEEE, pp. 1947–1950.

Sharma, Sugandha, Sean Aubin, and Chris Eliasmith (2016). "Large-scale cognitive model design using the Nengo neural simulator". In: *Biologically Inspired Cognitive Architectures* 17, pp. 86–100.

Sharp, T. and S. Furber (2013). "Correctness and performance of the SpiNNaker architecture". In: *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pp. 1–8. DOI: 10.1109/IJCNN.2013.6706988.

Sharp, Thomas, Rasmus Petersen, and Steve Furber (2014). "Real-time million-synapse simulation of rat barrel cortex." In: *Frontiers in neuroscience* 8, p. 131. DOI: 10.3389/fnins.2014.00131.

Sherman, S. Murray (2006). "Thalamus". In: *Scholarpedia* 1 (9), p. 1583.

Smolensky, Paul (1988). "On the proper treatment of connectionism". In: *Behavioral and brain sciences* 11 (01), pp. 1–23.

— (1990). "Tensor product variable binding and the representation of symbolic structures in connectionist systems". In: *Artificial intelligence* 46 (1-2), pp. 159–216.

Stewart, T.C. and C. Eliasmith (2014). "Large-Scale Synthesis of Functional Spiking Neural Circuits". In: *Proceedings of the IEEE* 102 (5), pp. 881–898. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2306061.

Stewart, Terrence C. (2012). *A Technical Overview of the Neural Engineering Framework*. Tech. rep. Centre for Theoretical Neuroscience, University of Waterloo.

Stewart, Terrence C., Xuan Choo, and Chris Eliasmith (2010). "Dynamic Behaviour of a Spiking Model of Action Selection in the Basal Ganglia". In: *10th International Conference on Cognitive Modeling*.

Stewart, Terrence C. and Chris Eliasmith (2009). "Spiking neurons and central executive control: The origin of the 50-millisecond cognitive cycle". In: *9th International Conference on Cognitive Modelling*.

Stewart, Terrence C and Chris Eliasmith (2011). "Neural cognitive modelling: A biologically constrained spiking neuron model of the Tower of Hanoi task". In: *the 33rd Annual Conference of the Cognitive Science Society*. Ed. by L. Carlson, C. Hlscher, and TF Shipley.

Stewart, Terrence C., Ashley Kleinhans, Andrew Mundy, and Jorg Conradt (2016). "Serendipitous Offline Learning in a Neuromorphic Robot". In: *Frontiers in Neurorobotics* 10 (1). ISSN: 1662-5218. DOI: 10.3389/fnbot.2016.00001.

Stewart, Terrence C., Yichuan Tang, and Chris Eliasmith (2011). "A biologically realistic cleanup memory: Autoassociation in spiking neurons". In: *Cognitive Systems Research* 12 (2), pp. 84–92. ISSN: 13890417. DOI: 10.1016/j.cogsys.2010.06.006.

Strata, Piergiorgio and Robin Harvey (1999). "Dale's principle". In: *Brain research bulletin* 50 (5), pp. 349–350.

Stromatias, Evangelos et al. (2015). "Scalable energy-efficient, low-latency implementations of trained spiking deep belief networks on SpiNNaker". In: *2015 International Joint Conference on Neural Networks (IJCNN)*. IEEE, pp. 1–8.

Tang, Yichuan and Chris Eliasmith (2010). "Deep networks for robust visual recognition". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 1055–1062.

The White House (2013). *Brain Initiative*.

Tripp, Bryan P (2015). "Surrogate population models for large-scale neural simulations". In: *Neural computation* 27 (6), pp. 1186–1222.

Tripp, Bryan and Chris Eliasmith (2016). "Function approximation in inhibitory networks". In: *Neural Networks* 77, pp. 95–106.

Turing, A M (1950). "Computing machinery and intelligence". In: *Mind* LIX (236), pp. 433–460. DOI: 10.1093/mind/LIX.236.433.

Verschure, Paul F.M.J. and Thomas Voegtlin (1998). "A bottom-up approach towards the acquisition, retention, and expression of sequential representations: Distributed adaptive control III". In: *Neural Networks* 11, pp. 1531–1549.

Webb, B (2001). "Can robots make good models of biological behaviour?" In: *The Behavioral and brain sciences* 24 (6), 1033–1050, discussion 1050–1094. ISSN: 0140-525X. DOI: 10.1017/S0140525X01000127.

Yavuz, Esin, James Turner, and Thomas Nowotny (2016). "GeNN: a code generation framework for accelerated brain simulations". In: *Scientific reports* 6.