Evolving Spiking Neural Networks for Spatio- and Spectro- Temporal Data Analysis: Models, Implementations, Applications

by

Nathan Scott

A thesis submitted to Auckland University of Technology
in fulfillment of the requirements for the degree of
Doctor of Philosophy

in the
School of Engineering, Computing, and Mathematical Sciences
Faculty of Design and Creative Technologies

2015

To S.O.

Contents

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements or references), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Signed: _____

Date: _____

List of Figures

List of Tables

Listings

Acknowledgements

During the duration this work, I have been lucky enough to have been supported by a great number of people.

To Prof. Nikola Kasabov, thank you for your constant support and guidance. I'm sure I am not the easiest student to supervise, so I should also thank you for your patience. I am greatly indebted to you for the huge number of opportunities I have been offered, which I know would not have been the case with any other supervisor. Благодаря ви много!

Prof. Giacomo Indiveri (INI, ETH & U. Zürich), grazie di tutto! It has been a pleasure working with you and your excellent research group. The Capo Caccia workshops have been the most intellectually stimulating challenges of my research so far, and I am grateful for the chance to attend with your support. I look forward to working with you in the future.

Prof. Zeng-Guang Hou (CASIA) 谢谢您的帮助. Thank you for kindly hosting me at your institution at such an early stage in my studies. Living in Beijing was a unique experience, and I greatly appreciate your hospitality.

My utmost gratitude to Joyce D'Mello, the heart of KEDRI. Thank you for the advice and for your tireless efforts to make everything easier for us.

Dr. Stefan Marks (CoLab, AUT) Vielen, herzlichen dank! It has been an extremely enjoyable few years working in so many different environments and on so many different things with you. I look forward to further Jedi training in the Holodeck!

My labmates at KEDRI, especially Neelava Sengupta, Reggio Hartono, Vivienne Breen, Josafaith Israel Espinosa-Ramos, Dr. Sripirakas Sakthithasan, and Dr. Paul Davidson: thank you for endless interesting conversations and for your help when an experiment went up in flames.

To those I have worked with outside KEDRI over these years, particularly Caroyln McNabb of UoA, Mahmoud Mahmoud of AUT's IRASR, and Dr. Simon Davidson, Prof. Steve Furber, and Jamie Knight of the SpiNNaker group at U. Manchester's APT, thank you for taking the time to share your work with me and including me in such interesting areas of research.

To my parents Clare and Doug Scott, my friends, especially Dr. Adam Reeve, Donna Reeve, Warren Nicholson, Chris Ryan, Ksenia Kovaleva, Andrew Smith,

and Anna DeKlerk, thank you for your help, your friendship, for putting up with me during this process, and for sticking around when I disappeared behind my computer.

Finally, thanks to my wonderful partner Sarah Ohlemacher, who is presently experiencing the same challenges in her own PhD studies. Thank you for your unwavering support, and understanding of the unique stresses and pressures such a project brings. I am sure yours will go more easily than this.

I am sure to have forgotten people here. To them I apologise. Your presence on or absence from this list is not a reflection on the level to which I am appreciative of your help. Thank you all.

Funding: This work was supported financially by the Auckland University of Technology Vice Chancellor's Doctoral Scholarship, and the Knowledge Engineering and Discovery Research Institute. Travel support was also provided by the Chinese Academy of Sciences Institute of Automation, the IEEE Computational Intelligence Society, the Convergent Science Network, and Zhejiang University. Some research was performed with funding from the New Zealand Ministry of Business, Innovation and Enterprise (MBIE) through a Strategic Partnership New Zealand-China Grant, and from the Auckland University of Technology's Strategic Research Investment Fund INTELLECTE Grant.

Publications

This thesis contains original material by the author published during the course of this study, in the following peer-reviewed papers:

1. Kasabov, N., Sengupta, N., **Scott, N. M.** (2016). From von Neumann, John Atanasoff and ABC to Neuromorphic Computation and the NeuCube Spatio-Temporal Data Machine. In *Proceedings of the 2016 IEEE Intelligent Systems Conference*. September 4–6. Sofia, Bulgaria. IEEE.
2. Kasabov, N., **Scott, N. M.**, Tu, E., Marks, S., Sengupta, N., Capecci, E., Othman, M., Doborjeh, M., Murli, N., Hartono, R., Espinosa-Ramos, J.I., Zhou, L., Alvi, F., Wang, G., Taylor, D., Feigin, V., Gulyaev, S., Mahmoud, M., Hou, Z.-G. and Yang, J. (2016). Evolving Spatio-Temporal Data Machines Based on the NeuCube Neuromorphic Framework: Design Methodology and Selected Applications. *Neural Networks, 78*, 1-14. Special Issue on "Neural Network Learning in Big Data". Elsevier. doi:10.1016/j.neunet.2015.09.011
3. Sengupta, N., **Scott, N. M.**, and Kasabov, N. (2015). Framework For Knowledge Driven Data Encoding For Brain Data Modelling Using Spiking Neural Network Architecture. In *Proceedings of the 5th International Conference on Fuzzy and Neural Computing*. 17–19 December 2015. Hyderabad, India. Springer. doi:10.1007/978-3-319-27212-2_9
4. **Scott, N. M.**, Mahmoud, M., Hartono, R., Gulyaev, S., and Kasabov, N. (2015). Feasibility analysis of using the NeuCube Spiking Neural Network Architecture for Dispersed Transients and Pulsar Detection. In *Proceedings of the 13th International Conference on Neuro-Computing and Evolving Intelligence*. February 19–20, Auckland, New Zealand.
5. Taylor, D., Chamberlain, J., Signal, N., **Scott, N. M.**, Kasabov, N., Capecci, E., Tu, E., Saywell, N., Chen, Y., Hu, J., and Hou, Z.-G. (2015). Brain-Computer Interfaces for Neuro Rehabilitation. In *Proceedings of the 13th International Conference on Neuro-Computing and Evolving Intelligence*. February 19–20, Auckland, New Zealand.
6. Marks, S., Estevez, J. E., and **Scott, N. M.** (2015). Immersive Visualisation of 3-Dimensional Neural Network Structures. In *Proceedings of the 13th International Conference on Neuro-Computing and Evolving Intelligence*. February 19–20, Auckland, New Zealand.
7. **Scott, N. M.**, and Kasabov, N. (2015). Feasibility of Implementing NeuCube on the SpiNNaker Neuromorphic Hardware Device. In *Proceedings of the 13th International Conference on Neuro-Computing and Evolving Intelligence*. February 19–20, Auckland, New Zealand.

8. Hu, J., Hou, Z.-G., Chen, Y., Kasabov, N., and **Scott, N. M.** (2014). EEG Based Classification of Upper-Limb ADL Using SNN for Active Robotic Rehabilitation. In *Proceedings of the 5th IEEE RAS/EMBS International Conference on Biomedical Robotics and Biomechatronics*. August 12–15, São Paulo, Brazil. IEEE. doi:10.1109/BIOROB.2014.6913811
9. Taylor, D., **Scott, N. M.**, Kasabov, N., Tu, E., Capecci, E., Saywell, N., Chen, Y., Hu, J., and Hou, Z.-G. (2014). Feasibility of the NeuCube SNN architecture for detecting motor execution and motor intention for use in BCI applications. In *Proceedings of the IEEE International Joint Conference on Neural Networks*. Beijing, China. IEEE. doi:10.1109/IJCNN.2014.6889936
10. Kasabov, N., Hu, J., Chen, Y., **Scott, N. M.**, and Turkova, Y. (2013). Spatio-temporal EEG data classification in the NeuCube 3D SNN Environment: Methodology and Examples. In *Proceedings of the 20th International Conference on Neural Information Processing*, 3–7 November 2013, Daegu, Korea. Springer. doi:10.1007/978-3-642-42051-1_9
11. **Scott, N. M.**, Kasabov, N., and Indiveri, G. (2013). NeuCube Neuromorphic Framework for Spatio-Temporal Brain Data and Its Python Implementation. In *Proceedings of the 20th International Conference on Neural Information Processing*, 3–7 November 2013, Daegu, Korea. Springer. doi:10.1007/978-3-642-42051-1_11

I have also disseminated aspects of its subject matter in the following forums:

12. **Scott, N. M.** and Kasabov, N. (2016) *Spiking Neural Networks: The Machine Learning Approach*. Invited Tutorial at the International Joint Conference on Neural Networks, World Congress on Computational Intelligence. Vancouver, Canada.
13. **Scott, N. M.** and Kasabov, N. (2015). *Spiking Neural Networks for Machine Learning and Predictive Data Modelling: Methods, Systems, Applications*. Invited Lecture and Tutorial Session at the IEEE Computational Intelligence Society Summer School on Neuromorphic and Cyborg Intelligent Systems. Hangzhou, China.
14. **Scott, N. M.** (2015). *Scientific Research Collaboration in the Asia-Pacific Region: Challenges and Opportunities*. Invited Talk for the Asia New Zealand Foundation. Auckland, New Zealand.
15. Kasabov, N., Modha, D., and **Scott, N. M.** (2015). *Spiking Neural Networks and Neuromorphic Data Machines*. Workshop Session at the INNS Big Data Conference 2015. San Francisco, USA.
16. Rast, A., Stokes, A., Rowley, A., Davies, S., Lester, D., Furber, S., Whatley, A., Luck, C., Iakymchuk, T., Ros, P.-M., Partzsch, J., Reza, A., Bi, S., Neil, D., Stefanini, F., Binas, J., **Scott, N. M.**, Waniek, N., Celiker, O., Isaacs, P., George, R., and Urgese, G. (2015). *AERIE-P: AER Intersystem Exchange Protocol*. Draft Protocol developed at the Convergent Science Network of Biomimetic and Biohybrid Systems Cognitive Neuromorphic Engineering Workshop, Capo Caccia, Sardinia, Italy.
17. **Scott, N. M.**, Indiveri, G., and Davidson, S. (2015). *Neucube on High Performance Neuromorphic Computers*. Invited Talk at the 13th International Conference on Neuro-Computing and Evolving Intelligence. Auckland, New Zealand.

18. Taylor, D., Chamberlain, J., Signal, N., **Scott, N. M.**, Kasabov, N., Capecci, E., Tu, E., Saywell, N., Chen, Y., Hu, J., and Hou, Z.-G. (2015). *Brain–Computer Interfaces for neuro rehabilitation.* Invited Talk at the 13th International Conference on Neuro-Computing and Evolving Intelligence. Auckland, New Zealand.
19. **Scott, N. M.**, Mahmoud, M., Hartono, R., Gulyaev, S., and Kasabov, N. (2015). *Neuromorphic Computing with NeuCube for Dispersed Transients and Pulsar Detection.* Invited Talk at the Computing for SKA (C4SKA) Colloquium. Auckland, New Zealand.
20. Theunissen, M. (2015, January 25). Brainy Robot Predicts Danger [Interview]. *Herald on Sunday.* Available online from http://www.nzherald.co.nz.
21. Beran, R. (2014, August 21). Using the Mind to Control Robots [Interview]. *Radio New Zealand.* Available online from http://www.radionz.co.nz.
22. **Scott, N. M.** (2014). *Spiking Neural Networks for Personalised and Predictive Medicine.* Invited Talk at the School of Pharmacy, Faculty of Medical and Health Sciences, University of Auckland. Auckland, New Zealand.
23. **Scott, N. M.** (2014). *Building the NeuCube: Tools and Techniques for In Silico Neuromorphic Simulation.* Invited Talk at the State Key Laboratory of Management and Control for Complex Systems, Institute for Automation, Chinese Academy of Sciences, Beijing, China.
24. Kasabov, N., and **Scott, N. M.** (2014). *Machine Learning and Predictive Modelling for Large Stream Data using Neuromorphic Computation.* Invited Talk at the International Neural Network Society 'Big Data and Neural Networks' Special Talks event. Beijing, China.
25. Kasabov, N., and **Scott, N. M.** (2014). *Spiking Neural Networks for Machine Learning and Predictive Data Modelling: Methods, Systems, Applications.* Tutorial Session at the International Joint Conference on Neural Networks, World Congress on Computational Intelligence. Beijing, China.
26. Kasabov, N., and **Scott, N. M.** (2014). *Predictive Data Modelling of Large and Fast Streams of Spatio- or Spectro- Temporal Data using Neuromorphic Computation.* Invited Talk at the Computing for Square Kilometre Array Workshop, Multicore World 2014. Auckland, New Zealand.
27. **Scott, N. M.** (2013). *Neuromorphic Computing for Spatio- and Spectro-Temporal Pattern Recognition of Neuroinformatics Data: Applications in Neurorehabilitation.* Invited Talk at the State Key Laboratory of Management and Control for Complex Systems, Institute for Automation, Chinese Academy of Sciences, Beijing, China.
28. **Scott, N. M.** (2012). *Pattern Association and Learning in Computational Models of Spiking Neural Networks.* Group Session Chair at the Convergent Science Network of Biomimetic and Biohybrid Systems Cognitive Neuromorphic Engineering Workshop, Capo Caccia, Sardinia, Italy.

Where these materials or discussions have contributed to this body of work they have been appropriately cited or otherwise acknowledged.

Abstract

Evolving Spiking Neural Networks for Spatio- and Spectro- Temporal Data Analysis: Models, Implementations, Applications

Nathan Scott

Doctor of Philosophy

School of Engineering, Computing, and Mathematical Sciences
Faculty of Design and Creative Technologies
Auckland University of Technology

Arguably the most significant challenge in modern machine learning regards how we address the complexities of Spatio- and Spectro-Temporal Data (SSTD); *i.e.*, data with some spatial, spectral, and temporal component. Addressing this issue is of vital importance to our understanding of the world around us.

Traditional machine learning techniques like the Support Vector Machine and Multi-Layer Perceptron struggle with the implicit representation of these characteristics. Typically, traditional ML abstracts away one or more of these components – and with it, a significant proportion of the information implicit in the relationships between place and time in the data. When we begin to look at brain data, seismic data, ecological data – in fact, any SSTD – this information is vital, and abstracting it is to destroy the data.

Instead, we can look to the brain for inspiration. The field of Spiking Neural Networks (SNN) – the mathematical-computational modelling of biological neural networks – provides a theoretical platform for the compact and integrated representation of spatial, spectral, and temporal characteristics in complex data. However, it is complex to design effective SNN which truly capture SSTD dynamics; indeed, this issue has yet to be adequately addressed in the present literature.

To this end, the NeuCube SNN framework has been abstractly established in recent works. Herein, the design and concrete implementation of systems based on this framework, and their practical application on SSTD is addressed. The NeuCube provides a framework for the processing of SSTD, including data encoding, reservoir computing, and classification. Additionally, immersive visualisation tools are introduced to facilitate the extraction of knowledge from the evolution of the model.

Firstly, a design methodology for the creation of NeuCube framework based SNN is introduced, including discussion of how to design reservoirs based on the implicit data structure, and encoding and output devices based on the data and selected application.

A complete software architecture and design philosophy for the implementation of such systems in software is then introduced. A concrete implementation developed in the Python simulator interface library PyNN is presented, including considerations for adaptive network structures and input mappings. This implementation has been developed for cross-platform, massively scalable simulation of NeuCube models.

Subsequently, the considerations for, and an implementation of, this architecture on a number of specialised computational platforms known as neuromorphic hardware is introduced. Neuromorphic hardware is a compact and power-efficient method of implementing SNN based on implementing the biophysical properties of neurons in dedicated circuits. Here is discussed preliminary work in implementing the NeuCube on FPGA, and neuromorphic VLSI systems such as the cxQuad. An implementation of the NeuCube on the SpiNNaker neuromorphic hardware device – a massively scalable digital computation platform – is provided and discussed.

Two primary appendices are attached to this thesis. Firstly, considerations for the design of NeuCube systems for spatio-temporal data are discussed, in the context of neuroinformatics. The most common neuroimaging tools (EEG and fMRI) are introduced here, and considerations for the design of NeuCube reservoirs to process such data is introduced, using the Talairach and Montreal Neurological Institute atlases. Empirical evidence of this system's effectivness on EEG based motor imagery is provided, where the NeuCube outperforms traditional ML techniques.

Secondly, considerations for the design of NeuCube systems in the context of spectro-temporal data are discussed, with a particular emphasis on radioastronomy. Introduced here is a conceptual mapping from spectral characteristics into the spatial structure of the NeuCube reservoir, which is a generalisable system. A proof-of-concept case study for the classification of complex spectro-temporal signals is presented, where it is shown that a NeuCube-based system can identify pulsar signals in synthetic radioastronomy data.

This thesis introduces generalisable design and implementation methodologies for SNN applied to complex SSTD, in the particular context of the NeuCube. Additionally, it provides some empirical evidence towards the efficacy of such methodologies for spatio-temporal and spectro-temporal data, in the context of neuroinformatics and radioastronomy respectively.

The principles now being discovered at work in the brain may provide, in the future, machines even more powerful than those we can at present foresee.

— J.Z. Young
(*Doubt and Certainty in Science: A Biologist's Reflections on the Brain*, 1960).

Chapter 1

Introduction

Arguably the most complex challenge in modern machine learning lies in the analysis, prediction, and classification of Spatio- and Spectro-Temporal Data (SSTD). That is, some data source which has its temporal component strongly coupled with its spatial or spectral components, or both. A vast proportion of data in the real world is composed of these properties. However, as yet, traditional machine learning practice struggles to address these issues effectively – even in isolation. This issue is compounded when we look at these data sources as an integrated signal. A vast amount of information is implicit in the relationships between the spatial, spectral, and temporal components of SSTD; information which is typically lost, abstracted away in order to mitigate the limitations of traditional machine learning techniques like the Support Vector Machine (SVM) and Multi-Layer Perceptron (MLP).

Consider a data source such as an Electroencephalographic (EEG) Brain Computer Interface (BCI). The strongly coupled nature of the spatial, spectral, and temporal components of the recorded brain signal is immediately apparent. The story of the subject's response to – for example – a visual stimulus, is told in the propagation and relative timing of activity from the eye, through the lateral geniculate nucleus, to the occipital cortex, and then on to the temporal and parietal lobes. To neglect these relationships is to neglect a significant portion of the information contained in the data. The question therefore then becomes – how do we model and interpret these dynamics in an appropriate way?

In short, we draw inspiration from that most powerful of adaptive learning devices: the human brain. Why should we be interested in the brain, when we are looking for improved techniques in information processing? The human brain excels like no

other system in its ability to learn from experience, operate in noisy and complex environments, and adapt when that environment changes.

A natural question then, is just how can we draw inspiration from brain processes in a computational context? Computational intelligence research addresses this through an area known as Spiking Neural Network (SNN) modelling. Here, we make some mathematical-computational model of the dynamics of biological neurons and networks thereof, thereby emulating – to some extent – the complex neurological dynamics which lead to the vast learning ability of the human brain. It is remarkably simple to model a single neuron. Their effectiveness lies not in their computational power as a single unit, but in the way a large group of them can communicate and collaborate with relatively simple inputs and outputs. They are a key example of an emergent system, wherein a computationally simple process at a large enough scale generates interesting and meaningful behaviours.

These networks, for all their apparent simplicity, are complex to design and optimise. Traditional techniques in this area have mitigated this issue by imposing some arbitrary structure on a network of neurons, which may or may not represent the internal dynamics required to deal with SSTD. To address this issue of arbitrary design, we can again take some cues from our environment. There is an implicit structure in SSTD by definition; it in some way represents a physical or environmental dynamic over time. With some informed decision-making, the structures inherent in SSTD can be modelled in the structure of a SNN, which will retain the vital interplay between the spatial or spectral components, and the temporal component.

To this end, the NeuCube framework was initially introduced in the work of Kasabov (2012b). In this, the NeuCube was abstractly defined. Herein concrete realisations of this framework, and design methodologies for the same will be introduced.

1.1 Research Questions

Inspired by this motivation, this thesis seeks to answer the following fundamental questions:

1. Can a specific SNN framework known as the NeuCube be used to model and interpret the dynamics of a system consisting of tightly coupled spatial, spectral, and temporal data components?
2. Is there a design methodology we can use to inform the development of NeuCube models?

With the assumption that those questions defined above are satisfied, the following supplementary questions are asked:

3. What considerations are there for an implementation of this system on:
   (a) Commodity computers,
   (b) Large scale clusters, or
   (c) Dedicated neuromorphic hardware?
4. Can we show some empirical evidence of a NeuCube model's effectiveness, when implemented on the above systems and designed utilising the newly established methodology, in the context of:
   (a) Spatio-Temporal data, and
   (b) Spectro-Temporal data?

1.2 Structure of this Thesis

The structure of this thesis is fairly straightforward. A schematic overview of its structure can be seen in Figure 1.1. In this section, we briefly discuss each section and its contributions to the thesis, in the order they appear in text.

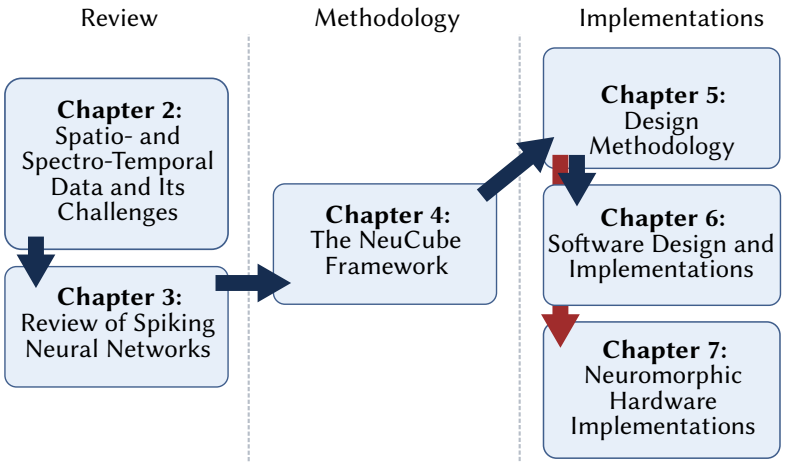


Figure 1.1: The structure of this thesis.

Chapter 2 – Spatio- and Spectro-Temporal Data: Challenges and Opportunities: Here, we briefly introduce the context of SSTD, and the particular challenges and opportunities it represents. This will be discussed in the context of existing machine learning systems, and their limitations which have motivated the development of this thesis.

Chapter 3 – Review of Spiking Neural Networks: This chapter introduces the existing work in the field of SNN, which provides the foundation for the primary contributions of this thesis. SNN is reviewed. Here, concepts of spike coding, contemporary models of SNN, learning algorithms, and simulation platforms for the same are introduced.

Chapter 4 – The NeuCube Framework: The chapter introduces the NeuCube framework as it existed in the literature prior to the work in thesis. It introduces the basic concepts of the different NeuCube components, and also discusses an immersive visualisation system which has been developed specifically for the NeuCube.

Chapter 5 – Design Methodology for SNN based on the the NeuCube Framework: Here a methodology for the development of NeuCube models is discussed. This introduces considerations for the design of encoding systems, reservoir topologies and parameters, output devices, and an overview of the influences our computational platform has on the model.

Chapter 6 –Software Design Methodology and Implementations: In this chapter, a general implementation framework for software realisations of the NeuCube, including a general design philosophy and software architecture, is introduced. An implementation of this framework is shown and discussed, developed in Python using the PyNN library. An alternative software architecture known as the NeuCube Core is also briefly introduced.

Chapter 7 – Neuromorphic Hardware Implementations: Here, a number of considerations for implementation of the NeuCube on neuromorphic hardware systems are discussed. An implementation of the NeuCube for the SpiNNaker neuromorphic device is presented. Considerations for analog- and FPGA-based implementations are discussed.

Chapter 8 – Conclusions: Here conclusions, a summary of contributions from the thesis, and suggestions on future works are presented. We close with some final thoughts. Subsequent to this section, the references, list of abbreviations, and Appendices can be found. Two key Appendices should be highlighted:

1. **Appendix A – Case Study in Spatio-Temporal Data:** This chapter discusses applications of the NeuCube for Spatio-Temporal data, in the context of a neuroinformatics task. An experiment which seeks to identify the NeuCube's potential efficacy in a BCI task is performed.

2. **Appendix B – Case Study in Spectro-Temporal Data:** This chapter identifies considerations for Spectro-Temporal data in the NeuCube. Additionally, a proof-of-concept experiment which seeks to identify the NeuCube's effectiveness in identifying radioastronomy signals is presented.

Chapter 2

Spatio- and Spectro-Temporal Data: Challenges and Opportunities

> Nature uses only the longest threads to weave her patterns, so that each small piece of her fabric reveals the organization of the entire tapestry.
>
> — Richard P. Feynman
> (*The Character of Physical Law*, 1964)

In the previous chapter Spatio- and Spectro-Temporal Data (SSTD) is mentioned as a particular challenge in modern machine learning. Its properties have prompted the development of the systems explored in this thesis. So, for clarity, what *is* SSTD, and why is it so challenging to deal with?

Firstly, we should define SSTD. In our context, we consider it as data which must contain:

1. A temporal component – *i.e.* that such data must change over time; and,
2. One or both of the following:
   (a) A Spectral component – *i.e.* that the data has some spectral or frequency based information; and/or,
   (b) A Spatial component – *i.e.* that the data has some physical representation, such as a multiple collection channels which are physically dispersed in some way.

The interplay between these components gives rise to the significant complexities of SSTD. Simultaneously, this interplay is the source of the significant theoretical advantages of such data. The two are inextricably linked.

It is, therefore, a difficult proposition to address these complexities in such a way as to not destroy the theoretical advantages they present. There are a number of additional characteristics of such data. In this section, we have identified three primary considerations. Each of these enumerated here is a contributing factor in traditional machine learning's inability to effectively represent SSTD.

A key issue here is the concept of what we have termed *representation*.

1. Representation

Primarily, it is meant that alternative machine learning techniques are not generally capable of adequately addressing these significant intrinsic properties in their representations of data; those of space and time. With such a temporal component, the data is continuous rather than discrete (Bogorny & Shekhar, 2010). Additionally, the spatial charateristics of the data can have some influence on each other.

Traditional techniques typically separate these two components; *i.e.*, they are analysed individually as a temporal stream, and as spatial data (Venkateswara, Govardhan & Chalapati, 2012). However, it is somewhat disingenuous to imply that this is spatio-temporal data analysis. These two components are inextricably linked; it is naïve to assume that the spatial characteristics of the data have no effect on the temporal, and vice versa.

Consider a simple example here. Say that we take telemetry from a moving car. The engine speed (RPM) over time, or, the temporal component, is meaningful in and of itself. Its spatial component (physical location on a road) is also meaningful in isolation. However, to model such a data source in this way discards much of the important contextual information in the data. If they are modelled together, we can infer some meaningful knowledge. A high engine speed over time will indicate that the vehicle is travelling at a high speed. In conjunction with our knowledge of its spatial context, we can – for example – judge whether this vehicle is exceeding the road speed limit near a school. This contextualisation is the core precept of SSTD data mining. The relationships within this data are both *significant* and *meaningful*.

Let us consider the temporal component in isolation. This by itself is often not represented effectively in traditional machine learning algorithms (Kasabov, Hu, Chen, Scott & Turkova, 2013). In this case, we typically use a concept called 'windowing', where a segment of a temporal sequence is separated out, and each temporal value in that sequence is treated as an individual static variable. The temporal nature of the data is implicitly ignored when the data is flattened in this way. It is akin

to compressing a video to a single static frame; clearly, information remains, but significant data can be abstracted away in favour of a simpler data representation.

This 'flattening' effect is acceptable in certain contexts. For basic image recognition in moving images, it may lose no significant information. These are, after all, merely static images shown in quick succession. However, when we attach some context like a human's interaction with those objects, this temporal – and indeed, spatial – data is significant and meaningful. This issue is even more pronounced when we consider more complex data sources like neuroinformatics recordings. The nature of how our brains process information is inherently a spatio-temporal process, as data propagates through the various specialised areas of the brain over time.

Most traditional machine learning algorithms can generally not effectively address the issues of temporal data. Additionally, these generally struggle to represent a single data source in multiple formats in the same model, and as a result, incorporation of the the relationships between the spatial and the temporal is difficult.

The issue of representation in SSTD is closely related to the fact that it generally represents some physical behaviour or quantity in the real world; the data has some inherent structure and with it, meaningful internal relationships.

2. Structure

Structure does not necessarily mean *spatial* structure – although that is certainly a component. By structure, we define that there are some inherent relationships within the data. Place and time are intrinsically linked within SSTD. This issue is heavily entwined with the issue of representation discussed above.

However, there is an important semantic difference here. Whereas previously we were discussing the issue of representing the data in the model, here we are identifying the issue of how the source of the data is represented in the data itself. This is a lower level of abstraction than the issue of representation. It is also one that traditional machine learning generally ignores (Andrienko, Malerba, May & Teisseire, 2006). There is an opportunity to improve the effectiveness of our systems by incorporating this low-level information. A better representation of the data generation paradigm in the model is advantageous in terms of that model's effectiveness in addressing the data (Lemm, Blankertz, Dickhaus & Müller, 2011).

3. Non-stationarity

Typically in SSTD there is some component of non-stationarity. The data, of course, changes over time as a consequence of being a temporal sequence. However, in the context of how such data is generally collected, there is the potential for some non-stationarity in the data. By this is meant some slow change of the data baseline over time.

A key example of this is in the case of neurorehabilitation. Neurorehabilitation is the process of restoring, minimising and compensating for functional deficits in people suffering from spinal or neurological insults. Typically, this takes the form of some physical rehabilitation (exercises) in conjunction with neurological feedback (Sastre-Garriga, Galán-Cartañá, Montalbán & Thompson, 2005).

In the case that this process is to be mediated by a robotic device (*cf.* Section A.2) or BCI, the control system must be trained on the subject's behaviour. As a natural result of this rehabilitation process, the subject will gradually improve at the task, whether through task-specific adaptation or through improvement of their physical capabilities. As they improve, there will be some baseline change in the behavioural data classified by the BCI device. This shift will impair the system's ability to classify accurately, and may as a result actively discourage further improvement of the subject due to negative feedback (D. Taylor et al., 2014). Equally, this non-stationarity could be induced by sensor degredation or seasonal effects, for example. Here, we need a system which can adapt to the changing baseline with no loss of accuracy.

2.1 Examples of Spatio- and Spectro-Temporal Data

With the understanding of what we define SSTD as, it should be contextualised. It is difficult to choose examples of SSTD as it is so common in the real world. Here, we introduce the background of SSTD recorded from two vastly different sources; the human brain, and space.

2.1.1 Neuroimaging

A considerable number of different neuroinformatics modalities exist. However, only a few of these have as-yet attained mainstream appeal and availability in the typical collection contexts of neuroinformatics data. This section will briefly introduce the most common of these technologies. Electroencephalography (EEG) has been

discussed in more depth there as it provides context for the case study introduced, and was the initial area of application for the NeuCube.

Two key categories of neuroimaging devices exist: invasive, and non-invasive. In the case of 'invasive' devices, these typically require some surgical intervention in order to place them. They are invasive in the sense that they are in some way inside the body; a typical example in electrophysiology is the use of a patch clamp for directly recording the action potentials of neurons. Such devices are costly and require highly specialised environments in order to be implanted and to operate effectively. Non-invasive devices, however, record signals from outside the body, and as a result avoid the issues with invasive data collection. Additionally they are generally cheaper and may not require specially trained personel to operate them.

Those readers familiar with this area will note that it is more precise to consider EEG as an electrophysiology technique, rather than a neuroimaging one. However, here these have been incorporated together as this better represents the general family of data sources we use in the neuroinformatics form of the NeuCube.

2.1.1.1 Electroencephalography

Electroencephalography, typically abbreviated as EEG, is a non-invasive form of electrophysiological monitoring (Niedermeyer & Da Silva, 2005). It generally uses electrodes placed on the outside of the scalp, which are arranged using the International 10-5, 10-10, or more commonly, 10-20 system. This placement system is discussed later. EEG represents the synchronous activity of a small volume of neurons as it reaches the scalp. Estimates suggest that each electrode represents the averaged activity of around 1 cm^3 of brain volume.

EEG provides us with a high-frequency signal, most commonly between 256 Hz and 512 Hz, up to a typical limit of $\approx$ 2000 Hz and in some extreme cases, 20 000 Hz. The caveat here is that as this is a surficial electrode reading, it is heavily influenced by the electrical activity of muscle activation on the head. Activities such as blinking eyes or chewing motions can cause significant artifacting of the EEG signal, as the electrical potentials evoked by these muscle movements are much stronger than the typical 10-100 μV potentials evoked by brain activity (Aurlien et al., 2004).

Readers interested in the technical details and applications of EEG are directed to the standard text in this area, Niedermeyer and Da Silva (2005).

EEG is an interesting data source, as it contains rich information about the temporal dynamics of the brain, and the signal responds rapidly to mental activity due to its relatively high sampling frequency. As a result, it is commonly used in medical diagnostic tasks, and is increasingly being used as a data source for BCI applications.

SNN have been used for EEG analysis, and have shown remarkable performance in comparison to other traditional methods for classification task. For example, an implementation of spiking neural networks for EEG classification of epileptic seizure detection was shown in Ghosh-Dastidar and Adeli (2007), and Ghosh-Dastidar and Adeli (2009). Their experiments resulted in high classification accuracy, approximately 90% correct. Buteneers, Schrauwen, Verstraeten and Stroobandt (2008) analysed rat EEG data using a reservoir computing approach known as an echo state network, for epileptic seizure detection in real-time, based on data from 4 EEG channels. The study claimed that performance was higher than the other four traditional methods in terms of detection time, which was around 85% accuracy in 0.5 seconds for seizure and 85% accuracy in 3 seconds for tonic-seizure. However, this study is limited in its credibility due to the use of rat EEG data, acquired from only 4 channels and with a foreknowledge of the key frequencies for detecting seisure (8, 16, and 24 Hz).

It has been shown in Nuntalid, Dhoble and Kasabov (2011) that traditional classifiers do not perform optimally on raw EEG data with no preprocessing. However, when suitably encoded (in that case with the Ben's Spiker Algorithm) and passed through an SNN reservoir, those same classifers work especially well. This suggests that SNN, with their inherent temporal dependency, are appropriate for use on this temporal data. There is therefore an opportunity for us to apply the NeuCube framework with its meaningful reservoir structure and adaptive learning to this task.

EEG has been classified perviously in the literature using traditional techniques such Support Vector Machines (Garcia, Ebrahimi & Vesin, 2003), non-negative tensor factorisation (Lee, Kim, Cichocki & Choi, 2007), and the Common Spatial Pattern algorithm (Tomioka & Dornhege, 2006) all focusing on the spectral domain, and multilayer perceptrons with the intent of using the Berlin Brain-Computer Interface tool for text entry (Müller et al., 2008). While these attempts have been effective to some degree, they have all required a significant amount of data preprocessing. In addition, none of these techniques adequately address the issues of spatial and temporal dependency in the data. The contention that raw EEG data can be accurately classified by SNN systems was confirmed in recent papers (J. Hu, Hou, Chen, Kasabov & Scott, 2014; Kasabov, Hu et al., 2013; D. Taylor et al., 2014) on this subject, utilising the architecture proposed in Appendix A.

Perhaps more importantly in this case, none of the existing classification technolgies adequately address (or indeed, address *at all*) the issue of learning in the subject. It is well known that repeated practice of a task will increase efficiency and adaptation to that task in a subject. This requires a classification tool which can adapt or change with the subject as they learn or rehabilitate. A traditional technique such as the SVM involves the creation of a classification tool which is, at the end of training, fixed. The system's adaptation or retraining requires it to be completely regenerated; there is typically no facility for iterative updating of its training data. In contrast, the NeuCube framework is able to adapt on-line, and is therefore more appropriate for application in such a context.

International 10-20 and 10-10 Systems

As mentioned, EEG electrodes are placed on the scalp using a standardised system known as the International 10-20 system. In fact, there are three of these systems, each with different resolution; the International 10-20, 10-10, and the least common, 10-5. The first of these was introduced in Herbert (1958), and comprises a set of standard locations for electrode placement based on proportional measurements between the inion and nasion of the skull. This location system is supported by all EEG devices, with the specific system generally defined by the number of electrodes present in the device. See Figure 2.1 for a visual intuition of this system.

2.1.1.2 Magnetic Resonance Imaging

Magnetic Resonance Imaging (MRI) is a non-invasive neuroimaging technique, which uses the interference our body provides to a strong magnetic field to generate a 3D image of the brain in 'slices' (Lauterbur, 1973). An oscillating magnetic field is first applied, which excites the hydrogen atoms present in the water in our body. These excited atoms then emit a characteristic signal. By redirecting the angle of the magnetic field, positional information about the source of these signals can be inferred, which provides us with a representation of the physical structure of the area in question. A number of these 'slices' can then be stacked together to form a 3D image of the brain, comprised of 3D pixels known as voxels. Different forms of Magnetic Resonance Imaging (MRI) exist; we are primarily interested in 'Functional' Magnetic Resonance Imaging (fMRI), but structural MRI and the derivative Diffusion Tensor Imaging (DTI) are also of interest. This introduction has described the basic concept of structural MRI.

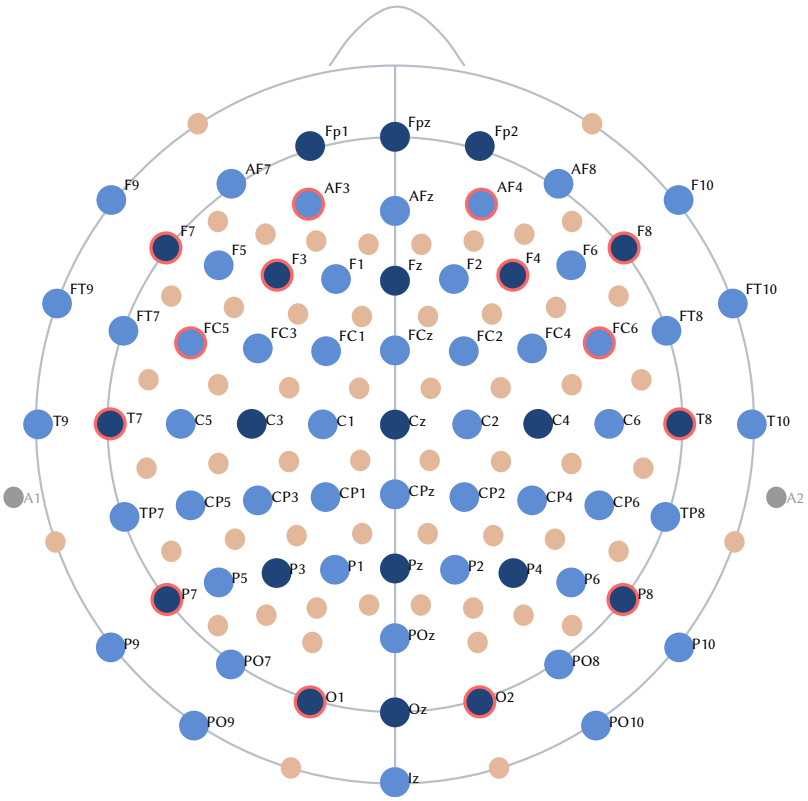


Figure 2.1: Scalp chart for the standardised EEG sensor positioning systems. Each coloured dot represents an electrode location on the scalp. Top of the page is the anterior direction (towards the nasion). Darker blue dots are International 10-20 standard locations. Lighter blue dots are International 10-10 standard locations. Orange dots are expanded International 10-5 locations. Red dotted rings around a location indicates that this is a sensor in the Emotiv Epoc device. Locations A1 and A2 are optional reference or ground electrodes attached near the ear. Figure inspired by Oostenveld and Praamstra (2001).

Functional MRI

Functional MRI, or 'Functional' Magnetic Resonance Imaging (fMRI), is an extension of structural MRI. While structural MRI is a static image of the brain, fMRI is a temporal sequence of these images. Typically, lower resolution structural MRI images are taken at a time interval (generally $\approx$ 500 ms). In this case, we do not look for the density of the water in the brain; we instead look for the Blood Oxygenation Level Dependent (BOLD) signal. This represents the amount of oxygenated blood in an area at the given time (Thulborn, Waterton, Matthews & Radda, 1982). This is typically correlated with the intensity of activity in that particular voxel, as more active neurons require a greater blood supply to support them. We are particularly interested in this technique in a neuroinformatics context as it provides a 3D view

of the activity in the entire brain over time, when given some task (Heeger & Ress, 2002). Particular caveats, however, are the low temporal resolution and the high cost of such a device.

The current state of the art in fMRI classification tools are firmly statistical techniques, and classical machine learning techniques. Recently, the first studies applying SNN to fMRI data have been performed, in Doborjeh, Capecci and Kasabov (2014b) and Doborjeh, Capecci and Kasabov (2014a), and also reported in Kasabov et al. (2015). These studies utilise the NeuCube framework and the design process introduced in this thesis, and in particular, the considerations for developing reservoir structures for neuroinformatics discussed later in this chapter.

In a number of studies, it is established that the Naïve Bayes and Support Vector Machine techniques are (relatively) superior when compared to other techniques such as k-Nearest-Neighbour (kNN) or linear discriminant analysis (Cox & Savoy, 2003; Ku, Gretton, Macke & Logothetis, 2008; Misaki, Kim, Bandettini & Kriegeskorte, 2010; Mitchell et al., 2004; Naselaris, Prenger, Kay, Oliver & Gallant, 2009), and that nonlinear SVM are superior to linear SVM (LaConte, Strother, Cherkassky, Anderson & Hu, 2005)

Recently, the first paper on the use of raw fMRI data for classification has been published, in Misaki et al. (2010). By 'raw', in this sense it is meant that no 'spotlighting' or dimensionality reduction has been performed, and the data is unprocessed apart from standard warp and movement correction. This is also an advantage of the NeuCube architecture, in that theoretically no preprocessing has to be performed in order for the system to accurately learn and classify the data. Related to this, there has recently been a move away from single-location stimulus classification to multi-location stimulus classification (Formisano, de Martino & Valente, 2008).

It should be noted that while the BOLD contrast mechanism commonly used for fMRI analysis linearly reflects the neural activity at a given voxel, this is primarily thought to be showing the input and intracortical processing, not necessarily its spiking output (Logothetis, Pauls, Augath, Trinath & Oeltermann, 2001). Furthermore, the observations in that same paper suggest that general statistical analysis and thresholding techniques normally applied to the haemodynamic response of a brain to stimuli probably underestimates a great deal of actual neural activity, particularly where localisation is a factor. This contention is also supported in Formisano et al. (2008) and Naselaris et al. (2009). It is therefore suggested that the NeuCube approach

is superior with respect to this typical limitation, as there is no localisation effect or culling performed on the data.

Diffusion Tensor Imaging

Diffusion Tensor Imaging (DTI) is a further extension of structural MRI, which generates an image of water diffusion across physical tracts of the brain – *i.e.*, the axons (Mori & Zhang, 2006). Instead of a static magnetic field that is moved to generate the image, DTI uses two magnetic fields in opposition. In places where the magnetic fields are maximally coherent, there is no diffusion of water across the material. If there is some incoherence, this implies that the area of interest has changed its properties between magnetic pulses. From the duration, direction, and spacing between these magnetic pulses, the direction and magnitude of water diffusion can be measured, and from this, the size and direction of white matter tracts inferred (Alexander, Lee, Lazar & Field, 2007). We are interested in this as it is clearly a candidate for the meaningful initialisation of a reservoir connectome; if we are structuring the shape of the reservoir based on the properties of the subject's brain, it is logical to also take inspiration from their brain's connectome where possible.

2.2 Pulsars, Radiotelescope Transients, & Astrophysics

To address why we are interested in radioastronomy data as an example of SSTD, we must first provide some background. The reader is directed to the cited literature for comprehensive discussion of this area. Primarily, we discuss one narrow subset of this field; the search for, and anaylsis of, pulsars.

A pulsar ('pulsating star') is a super-dense, rotating neutron star, which emits a beam of electromagnetic radiation from its mangetic poles (Michel, 1982). This beam is relatively narrow in astronomical terms, and coupled with the rotation of the pulsar, causes a 'lighthouse effect' whereby this signal is periodic. See Figure 2.2 for a visual intuition of this behaviour. An additional property of a pulsar signal is its dispersion. By this, it is meant that the initial signal is dispersed over the journey from source to collection site by interstellar media, in much the same way that a prism diffracts light. The mechanism is different, but the principle is more or less the same. This dispersion creates a characteristic inverse-quadratic shaped signal, with lower frequencies arriving later. The degree to which this is dispersed is proportional to the distance travelled by the signal, *i.e.* how far away the pulsar is from Earth. A single signal of this type is considered a *transient*, and its source is

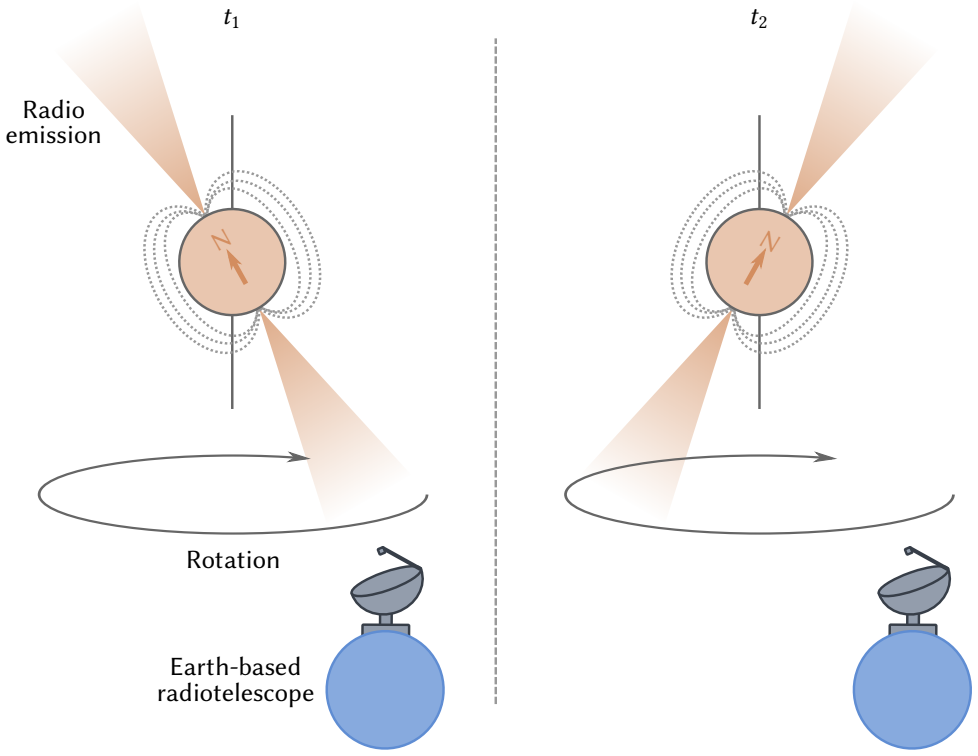


Figure 2.2: An intuition of a pulsar and its rotation affecting an earth-based radiotelescope at two time points. At time t_1 we see that the radio emission (orange cone) of the pulsar is directed at our radiotelescope's collection device. At t_2 we see that the pulsar has rotated around its axis and the radio emission is no longer directed at our collection device, causing the characteristic 'lighthouse effect'. Note that the axes of rotation and radio emission are not the same. The direction of radio emission is aligned with the magnetic poles of the pulsar. Here, the dotted lines denote this magnetic field.

unknown. It is theorised that these may be the result of a supernova explosion, but as yet, radioastronomy is not yet capable of localising and identifying these transients.

In fact, it is this periodicity which is the most interesting characteristic of a pulsar. A certain subset of these stars, known as milli- or micro-second pulsars, have such a regular periodicity that they are a more accurate timekeeper than an atomic clock (Backer, Kulkarni, Heiles, Davis & Goss, 1982; J. H. Taylor, 1991).

The regularity of the signal period is extremely useful in astrophysics. Any changes in its period are significant, particularly when the changes themselves have a periodicity. Here, we consider a pulsar with a companion star (*i.e.* two stars orbiting each other). If this companion star is of sufficent density, a gravitional lensing will occur when the pulsar is occluded by it, ostensibly changing the period of the pulsar. The magnitude of this change will be proportional to the two stars relative positions in

orbit. Of particular interest here is a pulsar with a companion black hole; due to the black hole's immense density, this will cause the most significant alteration to the pulsar's periodicity (Lyne et al., 2004). Additionally, by analysing and comparing the properties of similar pulsars with ordinary companions and pulsars with a black hole companion, it may be possible to shed some light on the properties of the black hole itself – currently a defiant challenge to astrophysicists. The interested reader is directed to the discussions in Backer and Hellings (1986), J. H. Taylor and Weisberg (1989) and the more recent papers by Angélil, Saha and Merritt (2010) and Deneva, Cordes and Lazio (2009).

The introduction of a new project which will vastly increase the number of pulsars identified is nearing completion; the Square Kilometre Array.

2.2.1 The Square Kilometre Array Project

The Square Kilometre Array (SKA)[1] is an international project to develop the world's largest radiotelescope array, totalling one square kilometre of collection area (Dewdney, Turner, Braun, Juande & Tan, 2015). Once completed, it will be approximately fifty times more sensitive than any other radiotelescope in operation, and require a data transmission capacity of one exabyte per day, around ten times larger than the total per-day bandwidth usage of the current Internet. As a result, it is spurring development in a number of areas of computation, including machine learning.

The SKA is intended to identify a total of around 2,000,000 new pulsars. Their identification in the data is a costly and laborious process. A recently concluded survey (one of the largest to date) identified a total of around 3,000 pulsars from 35,000 candidates, and took a team of graduate students nearly three years to complete (B. Stappers, personal communication, April 2014). The typical class ratio of pulsar : non-pulsar in a candidate signal is between 1 : 10,000-12,000. Clearly, at the scale of the SKA, it is no longer feasible to perform this search activity by hand. There is a pressing need for an automated method of candidate selection.

However, it is also a difficult task for machine learning to approach. This data has a number of characteristics which make it a significant challenge: most particularly, the scale, nonstationarity, noise level, and ambiguity of the data. This will be discussed further in Section B.2. These properties make the identification of pulsars in radioastronomy data an interesting application for the NeuCube, as these are similar to properties of neuroinformatics data we have already explored (*cf.* Appendix A).

[1] https://www.skatelescope.org/

2.2.2 Radio Frequency Interference Mitigation

Automated radio frequency interference mitigation technologies are also a key area of research in modern radioastronomy, and will be an important component in the effectiveness of the SKA. A Radio Frequency Interference (RFI) incident is some unwanted interference in the signal recorded by a radiotelescope; typically, this is caused by devices emitting radio waves, such as cellular telephones or aircraft radar. For this reason, the SKA will be located in 'radio quiet' areas of the desert. It is important to be able to identify these incidents quickly, so that mitigation techniques can be applied before the recording is spoiled or lost.

State of the art RFI identification at present is statistical in nature. The primary technique used is relatively straightforward spectral kurtosis. This compares the distribution of channel intensities in a Fourier-transformed signal of the different sensors in a radiotelescope Antoni (2006), Nita and Gary (2010). Effectively, this forms an analytic measure of the skewedness of the distribution, and with it, whether the signal is dispersed or not. Depending on the distribution, it can be inferred whether a signal is terrestrial in nature (*i.e.* an RFI event) or extra-terrestrial (*i.e.* a real signal).

Unfortunately, in the case of the SKA, this technique cannot be applied. It relies on the radiotelescope to be a multisensor array; *i.e.* that there are multiple sensors across a small collection area. The telescopes to be built in the SKA are single sensor. As a result, a picture of the skewedness of the signal cannot be developed, as there is no comparison possible between signals. An alternative technique is therefore sought for RFI identification.

2.3 Short-Term Fourier Transform and Alternate Representations of SSTD

It is possible in some cases to represent SSTD in an alternative form, which may represent the spatial and spectral dynamics of the data in a more compact way than in their raw form. One such example of this type of representation is the Short-Term Fourier Transform (STFT). Here, we perform a number of Discrete Fourier Transformations on small segments of an input signal, and treat these as the input vectors. We are therefore to an extent windowing the input data – although, in such a way that the direct relationship between the spectral change over time is still effectively represented.

This representation has not been addressed in the literature relating to the NeuCube at present, as emphasis has been primarily directed to the analysis of un-preprocessed data where possible. The impact of alternative data representations such as STFT when analysing SSTD should be addressed in future.

2.4 'Traditional' Techniques

In the previous discussion, 'traditional' machine learning was referred to. By the term 'traditional', we mean those techniques which are typically applied to datasets of this type, techniques which are well established in the literature. Here, we refer to learning systems like the MLP and SVM, and the more recent attempts at Recurrent Neural Networks (RNNs) and 'deep learning'. These techniques almost universally treat a temporal data stream as a static vector. Additionally, they are generally incapable of addressing multimodal data – in this case, data which contains both a temporal element and one of spatial or spectral components.

2.5 Addressing These Challenges

Current, the state of the art in SSTD analysis are typically statistical or standard machine learning models, such as the Support Vector Machine or real-valued RNN. These models do not take into account the inherent spatial and temporal dependencies within data such as EEG or fMRI, or general SSTD. In addition, these techniques commonly 'spotlight' areas of interest defined by some metric and discard the rest of the data (Lemm et al., 2011). For data such as fMRI, this can lead to vitally important dynamics being culled. The proposed NeuCube technique does not perform this type of dimensionality reduction, and is therefore theoretically superior in the sense that all of the data is retained (Kasabov et al., 2015). It is noted here that of course, this does mean that potentially spurious data is also retained. This factor, however, is handled through the unsupervised learning present in the reservoir computing stage of the framework.

There is a clear necessity for an effective evolving classification system identified in the literature. Applications as diverse as neurorehabilitation, brain computer interfaces, radioastronomy, and seismic prediction (among others) require accurate multimodal classification of complex SSTD. The current state of the art is reasonably effective, but there is significant room for improvement. Moreover, none of these models explicitly incorporate aspects of temporal dependence, spatial or spectral relationships within the data, or stochasticity, which are all implicit characteristics

of the data itself. In particular, none have incorporated these factors together into a cohesive tool for the classification and modelling of SSTD (Kasabov, Hu et al., 2013).

The context of this data and the limitations of the existing approches prompt the question – is there a better way? Can we address the challenges raised here in an attempt to exploit the considerable advantages of this type of data? There is a significant opportunity to improve upon the state of the art, and perhaps bring about a revolution in the classical thinking that has prevailed in the field to date.

We contend that there is, indeed, a better way. The following chapters will expand on, and justify, this contention, in both the context of SNN, and a specific SNN architecture known as the NeuCube. In the next chapter, we introduce the existing literature in SNN which will lead to a novel approach to the challenges established here.

Contributions of this Chapter

1. An overview and review of the challenges inherent in SSTD, and the limitations of existing approaches.
2. A novel characterisation of the challenges of SSTD.
3. The motivation for a new approach to these issues: Spiking Neural Networks and the NeuCube Framework.

Chapter 3

Review of Spiking Neural Networks

> By brain is meant, in the first instance, something more than the pink-grey jelly of the anatomist. It is, even to a scientist, the organ of imagination.
>
> — W. Grey Walter
> (*The Living Brain*, 1953)

Containing on the order of 10^{11} neurons and some 10^{14} connections, and consuming around 20 Watts, the human brain is arguably the world's most powerful and efficient computer. It is outclassed by orders of magnitude with regards to arithmetic computation, even when compared to a toy pocket calculator. However, its ability to learn from experience and function under uncertainty, at low power, and in the most extreme of environments dwarfs even the greatest current supercomputers.

In straightforward terms, Spiking Neural Networks (SNN) are an attempt to create biologically plausible computational and mathematical models of the inner workings of the brain at either a macro (outside the neuron) or micro (inside the neuron) level. SNN are typically considered to be the third generation of Artificial Neural Network (ANN) after the perceptron and the MLP. It could be argued that they are in fact, the fourth generation, after the modern forms of 'deep learning' architectures, as none of these models incorporate a temporal dependency. In addition, SNN are similar to biological neurons in that they communicate through discrete pulses of current known as 'spikes' (Maass, 1997), while previous generations of neural networks encoded information through continuous values and mean firing rates. See Figure 3.1 for a visual intuition of a spiking neuron.

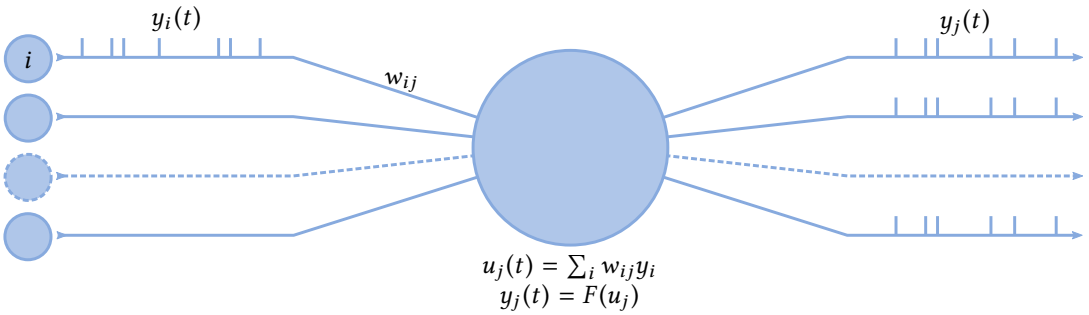


Figure 3.1: Illustration of a simplified spiking (temporally dependent) artificial neuron where: $y_i(t)$ is the input spike train from presynaptic neuron i over time; w_{ij} is the weight of an synapse between neurons i and j; $u_j(t)$ is the internal membrane voltage of neuron i over time; $F(u_j)$ is the neuron model (*e.g.* LIF, Izhikevich, *etc.*); and $y_j(t)$ is the output spike train from postsynaptic neuron j over time.

3.1 Why Spiking Neural Networks?

Spiking Neural Networks have a number of positive features, including compact representation of space and time; fast information processing; and time- and frequency-based information representation. They are therefore appropriate to address the issues raised in Chapter 2, in the analysis of spatio- and spectro-temporal data. Recently a number of novel SNN methods for spatio-temporal pattern recognition have been developed. Among them are two types of evolving SNN classifiers, including the deSNN by Dhoble, Nuntalid, Indiveri and Kasabov (2012), and SPAN by Mohemmed and Kasabov (2012), and related pilot applications for moving object recognition and simple EEG data classification.[1] This thesis proposes to further develop the existing research towards SSTD classification for a number of purposes, using the NeuCube framework recently proposed in Kasabov (2012b).

3.2 Biological Inspiration

Detail about the functionality of the human brain has been discussed in great depth in a number of textbooks. Those by Kandel and Schwartz (2000) and Bear, Connors and Paradiso (2007) are standard sources on the subject, from which the following brief summary is drawn.

Here, we will begin with a definition of a neuron. According to the literature, a number of features define a neuron: primarily, electrical excitability, and the presence of synapses, which are complex connectionist junctions that transmit signals to other

[1] `http://ncs.ethz.ch/projects/evospike`

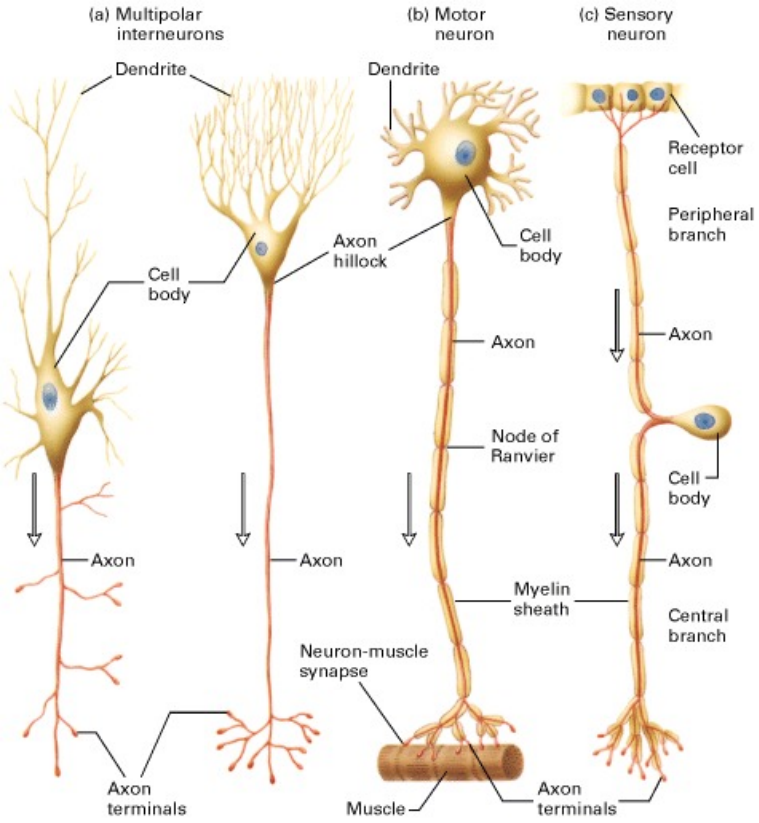


Figure 3.2: Diagram of typical mammalian neurons. Arrows indicate the direction of action potentials. (a) Multipolar interneurons (b) A motor neuron that innervates a muscle cell (c) A sensory neuron in which the axon branches just after it leaves the cell body. From Lodish et al. (2000).

cells. The body's neurons, plus the glial cells that give them structural and metabolic support, constitute the nervous system. Typically, the majority of neurons belong to the central nervous system, but some reside in peripheral ganglia, and others are situated in sensory organs such as the retina and cochlea. For the purposes of this thesis, we are primarily interested in the behaviour of neurons in cortex. See Figure 3.2 for a schematic illustration of a typical neuron.

A typical neuron is divided into three parts; the soma (cell body), dendrites, and axon. The soma (bulbous structure of Figure 3.2) is usually compact, while the axon and dendrites are filaments that protrude from it. Dendrites (thin branchlike structures off the soma in Figure 3.2) generally give rise to an abundance of branches which thin at each division, and extend their longest branches a few hundred micrometres from the soma. The axon (long thin structure towards the bottom of Figure 3.2) extends from the soma at a swelling called the axon hillock. Axons are capable of extending great distances, giving rise to hundreds of branches. Unlike dendrites, an axon usually maintains the same diameter as it extends. While a soma generally has

numerous dendrites, in no case does it ever extend more than one axon. These axons may be myelinated, or covered in a layer of a fatty substance known as myelin which acts as an electrical insulator.

Synaptic signals from other neurons are received by the soma and dendrites, while signals to other neurons are transmitted by the axon. A typical synapse is some area of contact between the axon of one neuron, and a dendrite or soma of another.

The synaptic signalling process, which is partly electrical and partly chemical, is the key to neural function. The electrical aspect depends on properties of the neuron's membrane. Every neuron is surrounded by a plasma membrane, a bilayer of lipid molecules with a number of different protein structures embedded within the plasma. This lipid bilayer is a strong electrical insulator. In these neurons, many of the protein structures embedded in the membrane are electrically active.

In particular, these electrically active components include ion channels that permit electrically charged ions to flow across the membrane, and ion pumps that actively transport ions from one side of the membrane to the other. Most ion channels are permeable only to specific types of ions (*e.g.* Calcium or Potassium). Some ion channels are voltage gated, meaning that they can be switched between open and closed states by altering the voltage difference across the membrane. Others are chemically gated, meaning that they can be switched between open and closed states by interactions with chemicals that diffuse through the extracellular fluid. The interactions between ion channels and ion pumps produce a voltage differential across the membrane, which provides a basis for electrical signal transmission between different parts of the membrane. The interactions between these ion currents are defined in the Nobel prize winning Hodgkin-Huxley model described in Section 3.3.1.

Neurons communicate by chemical and electrical synapses in a process known as synaptic transmission. The fundamental process that triggers synaptic transmission is the action potential, a propagating electrical signal that is generated by exploiting the electrically excitable membrane of the neuron. This is also known as a 'wave of depolarisation', or in our case, a 'spike'. The biological functions that cause this behaviour will be omitted here as this is beyond the scope of this thesis. Effectively, the net excitation received by a single neuron through its synapses over a short period of time reaches a firing threshold, the neuron generates a brief pulse called an action potential (which in SNN modelling is termed a 'spike'), that originates at the soma and propagates rapidly along the axon, activating synapses to other neurons. A synaptic transmission can be either excitatory or inhibitory depending on the type of synapse.

Different neurotransmitters and receptors are involved in excitatory and inhibitory synaptic transmissions respectively. Excitatory synapses release a transmitter called L-glutamate and increase the likelihood of the post-synaptic neuron triggering an action potential following stimulation. Inhibitory synapses on the other hand, release a neurotransmitter called γ-Aminobutyric Acid $(GABA)_A$ and decrease the likelihood of a post-synaptic potential. See the Hodgkin-Huxley model described in Section 3.3.1 and the Computational Neuro-Genetic Model (CNGM) in Section 3.6.3.3 for further details of these neurotransmitters.

Although action potentials can vary somewhat in duration, amplitude and shape, they are typically treated as identical stereotyped events in computational modelling. If the brief duration of an action potential (about 1 ms) is ignored, an action potential sequence, or 'spike train', can be characterized simply by a series of all-or-none point events in time (Gerstner & Kistler, 2002), generally modelled as δ-functions. The lengths of interspike intervals between two successive spikes in a spike train often vary, often randomly (Stein, Gossen & Jones, 2005). These interspike intervals give rise to the SNN's inherent temporal dependency. Spike timing precision in vivo is rarely below 5–10 ms (Butts et al., 2007; Desbordes, Jin, Alonso & Stanley, 2010, 2008; Marre, Yger, Davison & Frégnac, 2009) but as the timescales we work with in models such as these are arbitrary (in the sense that they are not the same as biological time), this is not a significant factor in the accuracy of our networks. It does however, suggest that coarser calculations than are the norm (updates every 1 ms) of network and neuron dynamics, or processing delays thereof, may be acceptable with no significant loss of accuracy.

3.3 Models of Spiking Neurons

As briefly discussed in the introductory section of this chapter, computational models of spiking neurons can typically be divided into two levels of abstraction: macro-, or micro-scopic models.

Microscopic spiking neurons refer to those which model the internal dynamics of the neuron with biological accuracy; that is, they calculate neuronal dynamics at the level of ion channels. A canonical example of this type of neuron is the Nobel Prize winning model introduced in Hodgkin and Huxley (1952).

However, for our purposes, macroscopic models are favourable for their lower computational costs and simpler implementation. With macroscopic models, we look at the neuron as a homogenous unit, or as a 'black box'. Here it is meant that we model

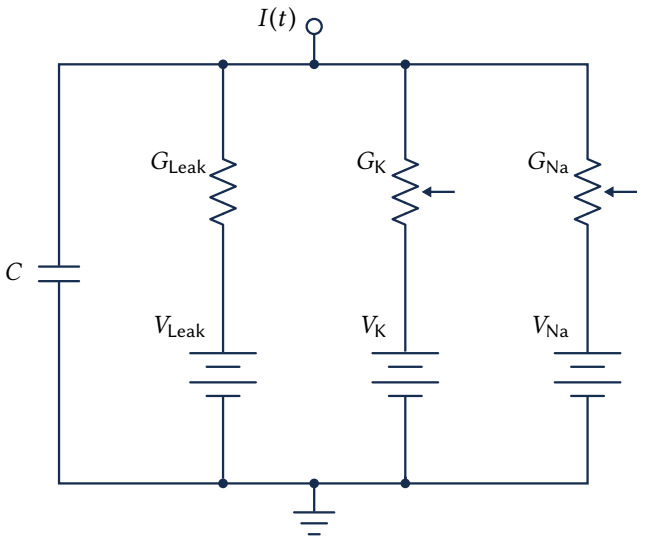


Figure 3.3: Example circuit schematic of the Hodgkin-Huxley model of neural behaviour. The semi-permeable cell membrane separates the interior of the cell from the extra-cellular liquid and thus acts as a capacitor. Ion movements through the cell membrane (in both directions) are modelled in the form of resistors. Inspired by Gerstner, Kistler, Naud and Paninski (2014).

the neuron at the level of its external dynamics and effects on neighbouring neurons, rather than at a great level of internal detail (Gerstner & Kistler, 2002). Therefore, this section will focus primarily on those macroscopic models which may be appropriate for implementation in this thesis. The Hodgkin-Huxley model will be introduced for context, and for its historical importance.

3.3.1 Hodgkin-Huxley

The Hodgkin-Huxely (HH) model of spiking neurons is one of the earliest proposed, first published in 1952. Drawn from empirical studies on the axons of a Giant Squid, it describes the influence conductances of ion channels have on the spike responses of the axon (Hodgkin & Huxley, 1952). Axons from a squid were selected due to their large size; at around 1 mm in diameter, they are one of the few axons visible to the naked eye. Contrast this with typical human axons at around 1 µm in diameter, or around one thousand times smaller (Debanne, Campanac, Bialowas, Carlier & Alcaraz, 2011). This large size was necessary as electrodes had to be inserted into the axon, in order to record the changes in electrical state experienced when neurons are active.

Three different ion currents were identified in the neuron; one sodium, one potassium, and one leak current. Voltage-dependent ion channels control the flow of ions across

the cell membrane. An active transport mechanism results in an electrical potential across the membrane, due to the ion concentration within the cell differing from that of the extra-cellular medium.

In their model, this membrane is described in terms of an electrical circuit with a capacitor in parallel with batteries and resistors. See Figure 3.3 for an example circuit schematic of this model. The current I at time t is split between that being stored in the capacitor and that passing through each of the ion channels. In formal terms, this is represented by

$$I(t) = I_{\text{cap}}(t) + \sum_k I_k(t) \tag{3.1}$$

From the definition of capacitance, $C = \frac{Q}{u}$ where C is capacitance, Q is charge, and u is the voltage across the capacitor, we are able to substitute $I_{\text{cap}}(t) = C\frac{\mathrm{d}u}{\mathrm{d}t}$, giving us

$$C\,\frac{\mathrm{d}u}{\mathrm{d}t} = -\sum_k I_k(t) + I(t) \tag{3.2}$$

As three ion channels are identified in this model, the sum above runs over all three, which are generally formulated as

$$\sum_k I_k(t) = G_{\text{Na}}\, m^3 h(u - V_{\text{Na}}) + G_{\text{K}}\, n^4 (u - V_{\text{K}}) + G_{\text{Leak}}\,(u - V_{\text{Leak}}) \tag{3.3}$$

where V_{Na}, V_{K} and V_{Leak} are constants known as 'reverse potentials'. The variables G_{Na} and G_{K} respectively represent the maximum conductance of the sodium and potassium channels, while the voltage-independent leak channel is represented as G_{Leak}. The variables m, n and h are gating variables whose dynamics are described in the differential equations

$$\begin{aligned}
\frac{m}{\mathrm{d}t} &= \alpha_m(u)(1 - m) - \beta_m(u)m \\
\frac{n}{\mathrm{d}t} &= \alpha_n(u)(1 - n) - \beta_n(u)n \\
\frac{h}{\mathrm{d}t} &= \alpha_h(u)(1 - h) - \beta_h(u)h
\end{aligned} \tag{3.4}$$

where m and h control the sodium channel, and n the potassium channel. Functions α_x and β_x, where $x \in \{m, n, h\}$ represent empirically determined voltage dynamics across capacitor u, are adjusted to simulate different types of neuron. See the parameters in Table 3.1 for values discovered in their experiments.

x	V_x (in mV)	G_x (in mS/cm^2)
Na	55	40
K	-77	35
Leak	-65	0.3

x	$\alpha_x(\frac{u}{\text{mV}})[\text{ms}^{-1}]$	$\beta_x(\frac{u}{\text{mV}})[\text{ms}^{-1}]$
n	$\frac{0.02(u-25)}{[1-e^{-(u-25)/9}]}$	$\frac{0.02(u-25)}{[1-e^{(u-25)/9}]}$
m	$\frac{0.182(u+35)}{[1-e^{-(u+35)/9}]}$	$\frac{-0.124(u+35)}{[1-e^{(u+35)/9}]}$
h	$0.25e^{-(v+90)}/12$	$\frac{0.25e^{(v+62)}/6}{e^{(v+90)}/12}$

Table 3.1: Hodgkin-Huxley model parameters, as reported in Gerstner, Kistler, Naud and Paninski (2014). The voltage scale has been shifted to have a resting potential of zero, and the membrane capacitance is $C = \frac{\mu\text{F}}{\text{cm}^3}$.

This model is computationally costly, and is generally used only when biophysical realism is vital. As previously mentioned, in engineering applications of SNN, we typically use macro-scale models instead. The following neuronal models are all macroscopic.

3.3.2 Izhikevich

The Izhikevich Model claims to combine the biological plausibility of the Hodgkin-Huxley with the lower computational complexity of Leaky Integrate-and-Fire type neurons (Izhikevich, 2003). Based on the theory of dynamical systems, the dynamics of this model are governed by two equations,

$$v' = 0.04v^2 + 5v + 140 - u + I \tag{3.5}$$

where v is the membrane potential of the neuron, and

$$u' = a(bv - u) \tag{3.6}$$

where u is a membrane recovery variable providing negative feedback for v; variables u and v, and parameters a, b, c and d, are dimensionless; and $' = \frac{\text{d}}{\text{d}t}$ where t is the time. Stimuli in the form of synaptic currents are represented by I.

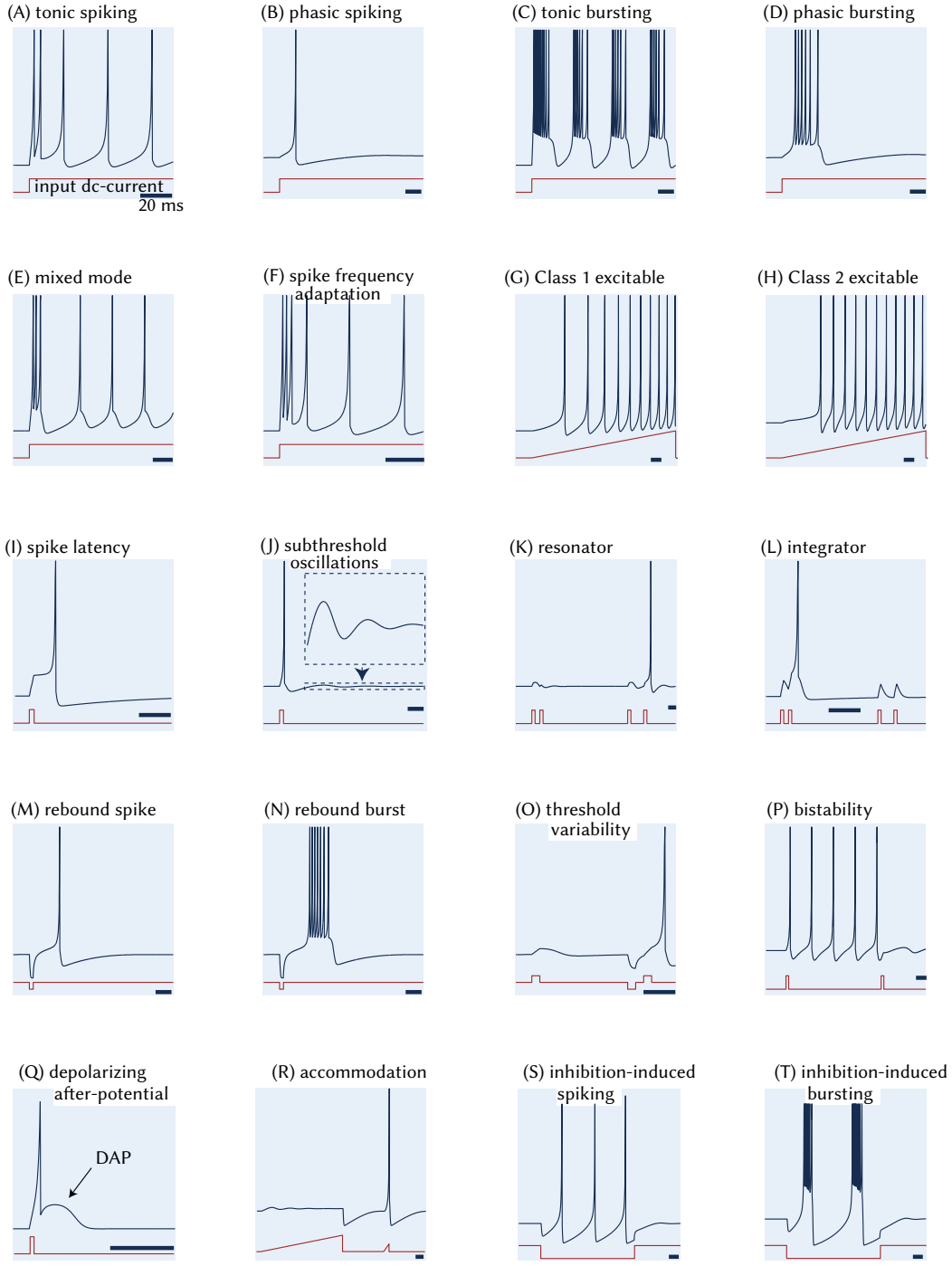


Figure 3.4: Illustration of the neuronal types it is possible to emulate using the Izhikevich neural model. In each sub-figure, the top (blue) trace is the membrane potential or internal voltage of the neuron, and the bottom (red) trace is the input (bias) current to the neuron. The solid bar to the bottom right of each sub-figure represents 20ms. Image recreated from Izhikevich (2004) and izhikevich.org.

When the membrane potential reaches the (fixed) threshold $\vartheta = 30\,\text{mV}$, the neuron spikes and resets u and v as

$$\text{if } v \geq 30\,\text{mV}, \text{ then } \begin{cases} v & \leftarrow c \\ u & \leftarrow u + d \end{cases} \tag{3.7}$$

Dependent on parameters a (decay rate of membrane potential), b (sensitivity of membrane recovery), c and d (reset values of v and u respectively), a huge variety of neuronal types can be modelled with relative ease. Izhikevich (2004) claims that all cortical neuron types can be modelled. See Figure 3.4 for a demonstration of this feature. Care must be taken to ensure that the parameters are selected such that the neuron type matches that required, as minor changes in these parameter values can have significant impacts on the model's behaviour.

3.3.3 Spike Response

Introduced in Gerstner and Kistler (2002), the Spike Response Model (SRM) is a generalisation of the Leaky Integrate-And-Fire (LIF) model, representing the state of a neuron with a single variable u. Different kernel functions are used to describe the effects of stimuli and presynaptic spikes on u, in addition to the shape of the actual postsynaptic spike and after-potential. As with the LIF, when variable u reaches the threshold ϑ from below, a spike is triggered. However, in contrast with LIF, this threshold ϑ does not need to be a fixed value, but can vary with the last spike time $\hat{t}_i$ of a neuron i. In practice, it is common for this behaviour to be utilised in the post-spike (refractory) period, increasing the threshold to suppress potential unwanted spikes.

The evolution of state variable u_i at time t, where $\hat{t}$ is the time of the last postsynaptic spike, can be modelled by

$$u_i(t) = \eta\left(t - \hat{t}_i\right) + \int_{-\infty}^{\infty} \kappa\left(t - \hat{t}_i, s\right) I(t-s)\,\mathrm{d}s \tag{3.8}$$

where $t_j^{(f)}$ are the firing times of presynaptic neurons j, w_{ij} represents the synaptic weight between neurons j and i, kernels η, ε and κ are response kernels, and the last term accounts for the effect of an external current $I(t)$ (Jolivet, Lewis & Gerstner, 2003). The integration process is characterised by the kernel κ, which is an additional external current. Kernel η defines the spike form and the post-spike membrane

potential if required. The choice of kernel κ is not restricted in the model, other than the fact that it inherits the time dependency of the overall model.

The spiking threshold ϑ can be constant or variable based on time. In the introductory paper, this is given as

$$\vartheta(t-\hat{t}) = \begin{cases} +\infty, & \text{if } (t-\hat{t}) \leq \gamma_{\text{refractory}} \\ \vartheta_0 + \vartheta_1 \exp(-\frac{t-\hat{t}}{\tau_\vartheta}), & \text{otherwise,} \end{cases} \tag{3.9}$$

Where $\gamma_{\text{refractory}}$ is a fixed refractory period; parameters ϑ_0, ϑ_1 and τ_ϑ are empirically chosen to yield an optimal fit to a test dataset (Jolivet et al., 2003).

3.3.4 Thorpe

A simplified form of LIF, in which each neuron in a network is allowed to emit a single spike at most, was formally proposed in Thorpe and Gautrais (1998). This formulation omits the post-synaptic potential leakage that is a common factor in the more complex spiking neural models. As a result, the spike response of a Thorpe neuron depends only on presynaptic spike times. It is capable of imposing precise timing on output spikes, although it is limited by the number of output spikes which can be generated for a single neuron.

Despite being limited to a single postsynaptic spike per neuron, this model is sufficiently effective to solve complex recognition tasks. This observation is consistent with studies of human sight, where it has been shown that processing of visual data requires only 150 ms (Thorpe, Fize & Marlot, 1996). In this short time, each neuron has the capability to emit very few spikes. As a result the Thorpe model is effective in these types of task, despite its simplicity.

The dynamics of this model are described by the time course of the postsynaptic potential $u_i(t)$ of a neuron i

$$u_i(t) = \begin{cases} 0 & \text{if } t > \hat{t}_i \\ \displaystyle\sum_{j \mid f(j)<t} w_{ij} m_i^{\phi(j)} & \text{if } t \leq \hat{t}_i \end{cases} \tag{3.10}$$

where w_{ij} is the weight of a presynaptic neuron j and $f(j)$ is the firing time of j, and $0 < m_i < 1$ is the modulation factor of the model. The function $\phi(j)$ represents the rank or index of the spike emitted by j. A rank of $\phi(j) = 0$ is assigned to the first spike of a given neuron if that spike is the first of all the presynaptic neurons,

and a rank of 1 if that spike is the second, and so on. In this way, the spikes of all presynaptic neurons are ranked and used in the calculation of u_i. Similar to most other neural models, a postsynaptic neuron i spikes when u_i reaches a threshold ϑ from below and resets to $u_i = u_r$, where u_r is generally zero, after this. The threshold $\vartheta = c\, u_{\max}$ is set to a fraction $0 < c < 1$ of $u_{\max}$, the maximum potential for that neuron (Schliebs & Kasabov, 2013). This model differs from normal multiply-firing neuronal models by the case in Equation 3.10. Here it is defined that if a neuron has previously spiked u_i is set to 0, thereby permanently suppressing any further spikes.

These simplifications of the general neural model allow for low computation cost when simulating large networks. As a result, the Thorpe model has been used in a number of complex domains, such as image and speech recognition (Thorpe, Guyonneau, Guilbaud, Allegraud & van Rullen, 2004) where it showed a reasonable level of effectiveness. This model is however, incompatible with the concept of precise spike timing for *multiple* spike output sequences. This is a constraint that means the Thorpe model is of limited used in the NeuCube paradigm, although it does see use in the Dynamic Evolving Spiking Neural Network Classifier (deSNN) classifier introduced in Section 3.6.3.2.

3.3.5 Leaky Integrate-and-Fire

The LIF neuron is one of the most popular models in the simulation of spiking neural networks. It was first described in the work of Louis Lapicque over a century ago (Lapicque, 1907), and discussed in a more modern context in Abbott (1999), and Brunel and Van Rossum (2007). Some biological plausibility is sacrificed for a more concise model, making LIF far less computationally costly than most other approaches.

It can be conceptualised as a simple electrical circuit containing a capacitor of capacitance C, and a resistor of resistance R. Both capacitance and resistance are constant. We represent the dynamics of a neuron i of this model by:

$$\tau_m \frac{\mathrm{d}u_i}{\mathrm{d}t} = -u_i(t) + RI_i^{\mathrm{syn}}(t) \tag{3.11}$$

where $\tau_m = RC$ is the membrane time constant of the neuron. When the membrane potential u_i reaches a threshold ϑ from below, a spike is emitted by the neuron and its potential is set to a reset potential u_r. From Gerstner and Kistler (2002) we then define the firing times of neuron i as

$$t_i^{(f)} : u_i\left(t^{(f)}\right) = \vartheta, f \in \{0, \ldots, n-1\} \tag{3.12}$$

where n is the number of spikes it emits. It is noted that the shape of the spike is not described in the basic LIF model; it only considers the firing times of the neurons. As a result, we can define a number of different shapes for the synaptic current. For example, Mohemmed, Schliebs, Kasabov and Matsuda (2011) uses an α-kernel, as

$$I_i^{\text{syn}}(t) = \sum_j w_{ij} \sum_f K_\alpha \left(t - t_j^{(f)}\right) \tag{3.13}$$

where $w_{ij} \in \mathbf{R}$ is the synaptic weight describing the strength of the connection between pre-synaptic neuron j and neuron i. The α-kernel used is defined as

$$K_\alpha(t) = e\, \tau_s^{-1}\, t\, e^{-t/\tau_s}\, \Theta(t) \tag{3.14}$$

where τ_s is the synaptic time constant (current decay rate) and $\Theta(t)$ refers to the Heaviside function, shown here as a function of time t

$$\Theta(t) = \begin{cases} 0, & \text{where } t < 0 \\ 1, & \text{where } t \geq 0 \end{cases} \tag{3.15}$$

which ensures that the kernel will equate to zero if the neuron has not emitted a spike.

We could define an exponential function to replace the α-kernel, such as that from Brette et al. (2007) and Vogels and Abbott (2005),

$$K_{\text{exp}} = e^{\frac{-t}{\tau_s}}\, \Theta(t) \tag{3.16}$$

where K_{exp} replaces K_α in the equation above, simplifying the model somewhat. The α-kernel is both more biologically plausible and easily controlled than this exponential shape.

LIF neurons are used in this thesis due to their relatively high level of biological similarity, low computational cost, and common presence across the simulation platforms used herein. Their dymanics are well understood and studied, and their behaviour can be compared across platforms to ensure that the various implementations of the NeuCube act as expected regardless of the simulation tool used.

3.3.6 Probabilistic Leaky Integrate-and-Fire

Synaptic transmission in vivo has been shown to be unreliable (Allen & Stevens, 1994; Hardingham & Larkman, 1998), and processing in neuronal systems thus unavoidably becomes a stochastic process (Tuckwell, 1988). Deterministic modelling of a stochastic process is impossible, and therefore, these models should be adapted to incorporate this probabilistic behaviour (Svirskis & Rinzel, 2000). The probability function used here is not specified in the literature, but typically a Gaussian function is utilised.

In a probabilistic model of the LIF neuron, first introduced in Kasabov (2010), three probabilistic parameters are added to the standard form.

- A probability $p_{c_{ij}}(t)$ that a spike emitted by a neuron n_j at time t through the connection c_{ij} between n_j and n_i will reach the neuron n_i. If $p = 0$, no connection and no spike propagation is possible between these two neurons. If $p = 1$, the connection exists and the probability of a spike reaching the postsynaptic neuron is 100%.
- A probability $p_{s_{ij}}(t)$ for the synapse s_{ij} to contribute to the total postsynaptic potential of neuron n_i after it has received a spike from n_j. If $p = 0$, no contibution will ever be made to the PSP of n_i. If $p = 1$, there will be a contribution made on every presynaptic spike.
- A probability $p_i(t)$ for the neuron n_i to emit an output spike at time t once the total internal potential reaches its threshold. If $p = 0$, neuron n_i will never spike. If $p = 1$, neuron n_i will spike every time it reaches its threshold.

The postsynaptic potential of the probabilistic spiking neuron n_i is now calculated as:

$$\mathrm{PSP}_i(t) = \sum_{p=t_0,t_1,\ldots,t} \sum_{j=1,2,\ldots,m} e_j \, g\left(p_{c_{ij}}(t-p)\right) f\left(p_{s_{ij}}(t-p)\right) w_{ij}(t) + \frac{1}{\tau(t-t_0)} \quad (3.17)$$

where e_j is 1 if a spike has been emitted from neuron n_j and 0 otherwise; $g\left(p_{c_{ij}}(t)\right)$ is 1 with a probability $p_{c_{ij}}(t)$ and 0 otherwise; $f\left(p_{s_{ij}}(t)\right)$ is 1 with probability $p_{s_{ij}}$ and 0 otherwise; t_0 is the time of the last spike emitted by n_i; and $\frac{1}{\tau(t-t_0)}$ is an additional term representing decay in the postynaptic potential. Here we have defined the model in terms of the postsynaptic potential, rather than its current as was the case

with the traditional LIF model; while it would be neater to cast this in terms of current, the voltage-based definition has been kept to keep consistency with the existing literature.

In the special case that all probabilities $p_{c_{ij}}, p_{s_{ij}}, p_i$ are set to 1, the model is simplified (*i.e.* there is no probabilistic element) and behaves as the standard Leaky-Integrate-and-Fire neuron. This neuron model is flexible and depending on the probabilistic parameters selected, can represent a number of biologically inspired behaviours. It is also suitable to integrate with the computational neurogenetic model discussed in Section 3.6.3.3 (Kasabov, 2012a).

In Schliebs, Nuntalid and Kasabov (2010) and Nuntalid et al. (2011) it is shown that replacing the deterministic LIF discussed above with a Probabilistic Leaky Integrate-And-Fire (pLIF) yields better separation and classification results. This may be a result of it mimicking biological behaviours known as "slow learning" (Amit & Fusi, 1994; Fusi, 2003). Slow Learning is a phenomenon where synaptic weight modification is moderated (or 'slowed') in order to reduce the significance of new input data on existing 'memories', thereby increasing the memory retention of the synapse and reducing potential impacts of any aberrant input.

Few other significant comprehensive probabilistic neural models have been proposed. By comprehensive, it is meant that those models incorporating considerations for both the axonal projections and the neuron itself. A simplified stochastic model was introduced in Svirskis and Rinzel (2000), which only modelled the membrane potential as a stochastic process.

3.3.7 Adaptive Exponential IAF

The Adaptive Exponential Integrate-And-Fire (AdEx) introduced in Brette and Gerstner (2005) is described by two equations – the first describing the dynamics of the membrane potential with an exponential voltage term, and the second which describes spiking threshold voltage adaptation. The combination of adaptation and exponential voltage dependence gives rise to the name AdEx. The AdEx model is shown in Naud, Marcille, Clopath and Gerstner (2008) to be capable of describing a number of known neuronal firing patterns (adapting, bursting, delayed spike initiation, initial bursting, fast spiking, regular spiking, *etc.*)

This model is described in Brette and Gerstner (2005) and Gerstner and Brette (2009) by two differential equations:

$$C\frac{\mathrm{d}v}{\mathrm{d}t} = -g_L(V - E_L) + g_L\Delta_T \exp\left(\frac{V - V_T}{\Delta_T}\right) - w + I \tag{3.18}$$

and

$$\tau_w\frac{\mathrm{d}w}{\mathrm{d}t} = a\,(V - E_L) - w \tag{3.19}$$

where: V is the membrane potential; w the adaptation variable; I the input current; C the membrane capacitance; g_L the leak conductance; E_L the leak reversal potential; V_T the threshold potential; Δ_T the slope factor; a the adaptation coupling parameter; and τ_w the adaptation time constant.

The exponential nonlinearity describes the process of spike generation and the upswing of the action potential. A spike occurs at the time t^f when the membrane potential V diverges towards infinity. In practice, integration of the model equations is usually stopped and the firing time t^f recorded when the membrane potential reaches a finite value (*e.g.* 0 mV or 30 mV). The downswing of the action potential is not described by the model but is generally implemented by a reset of the voltage to a fixed V_r, following the rule that:

$$\text{at } t = t^f \text{ reset } V \rightarrow V_r \tag{3.20}$$

Simultaneously, the adaptation value w is also altered by an amount b, following:

$$\text{at } t = t^f \text{ reset } w \rightarrow (w + b) \tag{3.21}$$

The AdEx is used in a number of neuromorphic hardware systems including Indiveri (2003), Indiveri and Horiuchi (2011), Moradi and Indiveri (2013) and Meier, Millner, Gr, Schemmel and Schwartz (2010). For further details, see Section 7.3.

3.4 Spike Information Coding

An open question in neuroscience regards the manner in which information is coded by neurons. The series of spikes (action potentials) that represents an event or message must have some structure in order for it to be meaningful to postsynaptic neurons. As mentioned in the previous section, if the brief duration of an action potential is ignored, a spike train is represented by a series of all-or-nothing events in time (Gerstner & Kistler, 2002). The difficulty then, is to extract meaningful

information from this spike train. While there is still significant debate regarding the form of neural encoding in the brain, two primary forms have been defined in the previous literature: rate and temporal (pulse) coding.

3.4.1 Rate Coding

Rate coding suggests that the mean firing rate of a neuron carries most or all of the significant information for a spiking event. A mean firing rate v is generally defined as the ratio of the average number of spikes n_{spikes} over a specific time interval t and this time interval itself, as $v = \frac{n_{\text{spikes}}}{t}$. This form of information coding is commonly used in older generation SNN, and has been identified in motor neural systems and sensory perception in animals (Adrian, 1926).

Rate coding as a concept is criticised, largely because of the comparably slow transmission of information between neurons (Dayan & Abbott, 2001). The minimum time possible for information to be extracted from the spike sequence is at least t_{min}, which is not the case for pulse coding. The extremely short response times of the brain to certain stimuli indicate that this form of neural encoding is too time-consuming to be effective. In particular, we note the case of human vision, where it has been shown that the brain can identify a visual stimulus in approximately 150 ms (Thorpe et al., 1996). With a moderate number of neural layers involved in this process, if every layer is made to wait at least time period $_{\text{min}}$ and given that each spike has a duration of 1 ms, this identification process would necessarily be slower. As a result, this method of encoding is inappropriate for representing sensory data, which is necessarily real-time. To address this issue, the concept of Temporal Encoding is used primarily in more recent works.

3.4.2 Temporal Coding

Temporal, or Pulse, coding assumes that the precise timing of the spikes in a sequence represents all or most of the significant information carried by that spike train (Bohte, 2004). A number of features, absent in rate coding, can be extracted from these time sequences. These include characteristics based on the second and higher statistical moments of the inter-spike interval probability distribution, spike randomness, or precisely timed groups of spikes (temporal patterns). As there is no absolute time reference in the nervous system, the information is carried either in terms of the relative timing or with respect to an 'ongoing brain oscillation', generally represented in simulations by an external clock. A growing body of literature supports the premise

of pulse coding. This includes theoretical studies of LIF models by Lestienne (1996), and empirical biological studies of rat cortices, both *in vitro* (Nawrot, Schnepel, Aertsen & Boucsein, 2009) and *in vivo* (Villa, Tetko, Hyland & Najem, 1999).

A popular form of temporal coding is known as time-to-first-spike, and was discussed by Thorpe et al. (1996). This encoding is based on the timing of the first spike after a reference signal (generally taken to be the beginning of a simulation period), and is inspired by human vision processing. As discussed above, it is argued that in this context, each neuron has the opportunity to emit only a few relevant spikes. As a result, it is suggested that earlier spikes carry a greater amount of information about the stimulus, and should be emphasised. The Thorpe model given in Section 3.3.4 is based on this principle.

3.5 Methods of Encoding Data into Spike Trains

A natural question at this point regards how we turn a real-valued input such as a time series from an EEG or seismograph into a train of spikes. A number of encoding schemes have been proposed in the literature. Here, we primarily focus on those providing precise spike times as their outputs, rather than rate-based encoding methods. One spike rate based method (the so-called BSA) will be introduced here as it has been applied successfully to EEG data in a previous study.

3.5.1 Temporal Difference (Threshold-Based)

The Threshold-Based Temporal Difference (TD) encoding algorithm is simple. In short, some threshold ϑ is defined. We calculate an output spike $s(t)$, such that

$$s(t) = \begin{cases} +, & \text{if } i(t) \geq \vartheta + i(t_{\text{last}}) \\ -, & \text{if } i(t) \leq \vartheta - i(t_{\text{last}}) \end{cases} \tag{3.22}$$

Where $i(t)$ is the input value at the current time and $i(t_{\text{last}})$ is the previous time point, and $+, -$ represent excitatory and inhibitory spikes respectively. In the case that neither of these conditions are met, no spike is emitted. The pseudocode of this method is given in Listing 3.1. See Figure 3.5 for a visual intuition of this process.

```
last_value = 0
for (i = 1 to size(input))
  current_value = input[i]
  if (current_value >= threshold + last_value)
    output_excitatory(i)
```

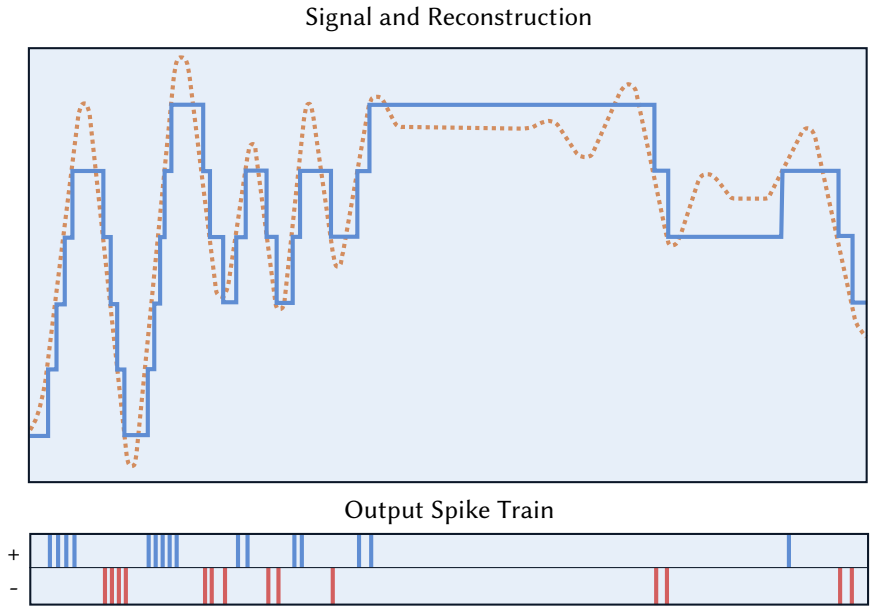


Figure 3.5: Illustration of the Threshold-Based Temporal Difference spike encoding scheme. In the upper figure, a real-valued input stream (orange dotted line) and its reconstruction (solid blue line) is converted to a series of excitatory (positive, blue) and inhibitory (negative, red) spikes, shown in the lower figure.

```
  else if (current_value <= threshold - last_value)
    output_inhibitory(i)
  end if
  last_value = current_value
end for
```

Listing 3.1: Pseudocode implementation of the Temporal Difference Encoding Algorithm.

Despite its relative simplicity, it has been established that this type of encoding is biologically plausible, particularly with regards to the encoding of visual stimuli (K. Kim & Rieke, 2001; Kuang, Poletti, Victor & Rucci, 2012; Oesch & Diamond, 2011; Rieke, 2001). TD is also appropriate for implementation in neuromorphic hardware systems (Freeman, Rizzo & Fried, 2011; Lichtsteiner, Posch & Delbruck, 2006).

Temporal Contrast & AER: The Temporal Difference algorithm is sometimes incorrectly conflated with Address-Event Representation (AER) due to the use of the conceptually similar Threshold-Based Temporal Contrast (TC) algorithm and AER use in the Dynamic Vision Sensor (DVS) (*cf.* Section 7.1.5.1). AER is in fact a *communication protocol*, not a spike encoding scheme. Some prior literature on the NeuCube including Capecci, Kasabov and Wang (2015), Doborjeh et al. (2014a), Kasabov and Capecci (2015), Kasabov, Hu et al. (2013), Scott, Kasabov and Indiveri

(2013) and Schliebs, Capecci and Kasabov (2013) carries this inaccuracy. In cases where these papers have referred to the use of 'AER Encoding', the Temporal Difference spike encoding algorithm is meant. See Section 6.3.3.1 for details of the AER communication protocol.

3.5.2 Population Encoding

Population Encoding, or, more properly, Rank-Order Population Encoding, is an extension of the rank order importance system in the work of Thorpe and Gautrais (1998). Using it, we can map a vector of real-valued elements into a spike sequence through the use of relative spike delays. An implementation based on Gaussian receptive fields was introduced Bohte, Kok and Poutre (2002). Receptive fields allow the encoding of continuous values through a collection of neurons with overlapping spike sensitivity profiles. Each input variable (channel) is encoded independently by a group of M one-dimensional fields. For each variable n we define an interval $\left[I^n_{\min}, I^n_{\max}\right]$. The Gaussian shaped receptive field of a neuron i is then given by its centre μ_i, where

$$\mu_i = I^n_{\min} + \frac{2i-3}{2} \times \frac{I^n_{\max} - I^n_{\min}}{M-2} \tag{3.23}$$

and its width σ, where

$$\sigma = \frac{1}{\beta} \times \frac{I^n_{\max} - I^n_{\min}}{M-2} \tag{3.24}$$

ensuring that $1 \leq \beta \leq 2$. Parameter β controls the width of each receptive field. In effect, we translate the relative similarity of the real valued input and its representative neurons to spike time delays for those neurons. The neuron representing the value closest to our input value spikes first, followed by the next most similar, and so on. See Figure 3.6 for a visual intuition of this method.

3.5.3 Ben's Spiker Algorithm

Ben's Spiker Algorithm (BSA), first introduced by Schrauwen and van Campenhout (2003) is an extension of the Hough Spiker Algorithm spike encoding scheme introduced in Hough, de Garis, Korkin, Gers and Nawa (1999).

The key benefit of using BSA is that the frequency and amplitude features are smoother in comparison to HSA. Due to the smoother threshold optimization curve, it is also less susceptible to changes in the filter and the threshold. Studies have shown that this method offers an improvement of 10 dB-15 dB Signal-to-Noise Ratio

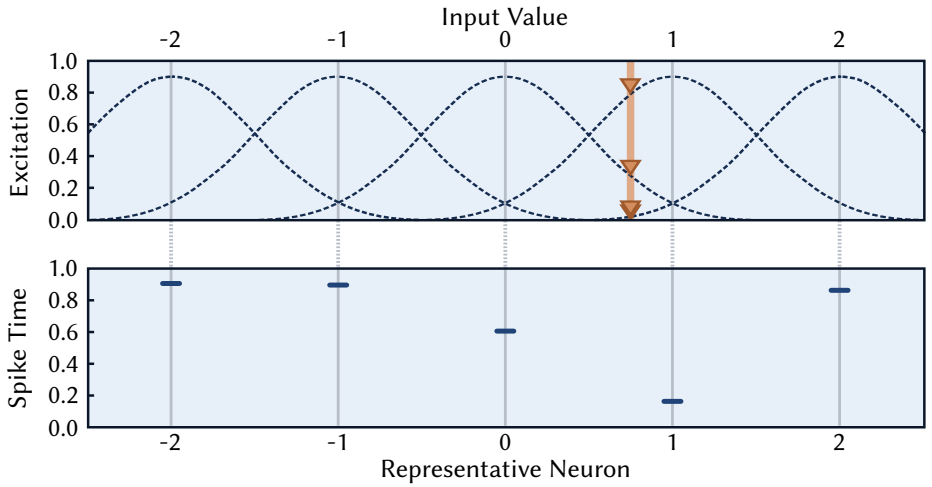


Figure 3.6: Illustration of the Rank-Order Population spike encoding scheme. For an input value $v = 0.75$ (shown here as a broad orange line in the top figure), the intersections with each Gaussian receptive field is shown, which are transformed into relative spike delays for the representative population of neurons. Here, the neuron representing the value '1' spikes at 0.2 s as it represents the proportionally closest value. Similarly, the neuron representing the value '0' spikes next, at time 0.6 s, and neurons representing values '2' and '-1' spike proportionally later. In this way, we can represent a continuous value as a discrete series of encoded spikes.

(SNR) over the HSA spike encoding scheme. See Figure 3.7 for a visual intuition of this encoding scheme.

According to Schrauwen and van Campenhout (2003), the stimulus s is estimated from the spike train

$$s_{\text{est}} = (h \times x)(t) = \int_{-\infty}^{+\infty} x(t-\tau)\, h(\tau)\, \mathrm{d}\tau = \sum_{k=1}^{N} h(t - t_k) \tag{3.25}$$

Where t_k is the neuron's firing time, $h(t)$ is the linear filter's impulse response, and $x(t)$ is the spiking behaviour of the neuron which is calculated as

$$x(t) = \sum_{k=1}^{N} \sigma(t - t_k) \tag{3.26}$$

The basic algorithm to calculate a spike train from a given input using Ben's Spiker Algorithm (BSA) is given here as pseudocode.

```
# i and j are the indexes of samples at time t_i and t_j respectively
for (i = 1 to size(input))
  error_1 = 0
```

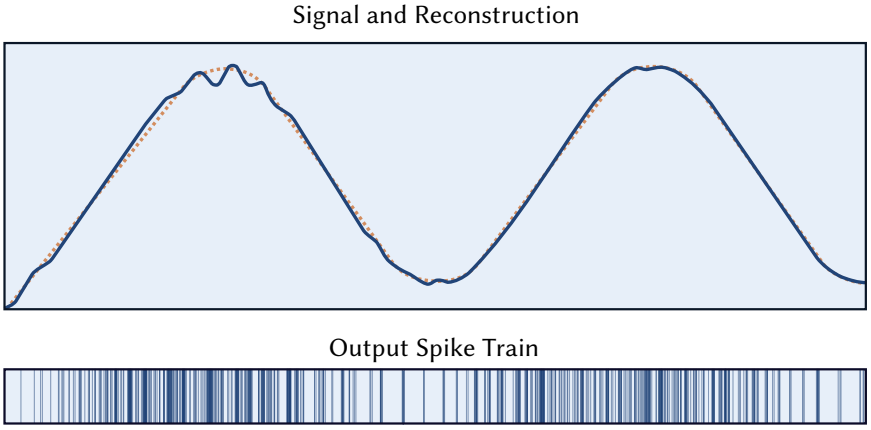


Figure 3.7: Illustration of the Ben's Spiker Algorithm encoding scheme. In the upper figure, the original signal (orange dotted line) and the signal reconstruction (solid blue line) after it has been converted to a train of spikes are overlayed. The output spike train generated by BSA is shown in the lower figure.

```
  error_2 = 0
  for (j = 1 to size(filter))
    if (i + j - 1 <= size(input))
      error_1 += abs(input(i_j - 1) - filter(j))
      error_2 += abs(input(i_j - 1))
    end if
  end for

  if (error_1 <= (error_2 - threshold))
    output(i) = 1

    for (j = 1 to size(filter))
      if (i + j - 1 <= size(input))
        input(i + j - 1) -= filter(j)
      end if
    end for
  else
    output(i) = 0
  end if
end for
```

Listing 3.2: Pseudocode implementation of Ben's Spiker Encoding Algorithm.

BSA was used successfully for the encoding of EEG-based brain data in Nuntalid et al. (2011). Here, it was hypothesised that as EEG is in the spectral domain, BSA (which is designed for sound data) could equally be applied. This contention proved accurate, with EEG spike train encoding using BSA shown to be highly effective. It has yet to be studied in the context of the NeuCube, but its effectiveness here does suggest that it is a viable area of future exploration.

3.5.4 Knowledge Driven Data Encoding Method

Introduced in Sengupta, Scott and Kasabov (2015), the Knowledge Driven Data Encoding Method is primarily designed for the encoding of brain data into trains of spikes.

The process of transforming neurologically defined events to a signal can be defined as a reverse encoding or decoding method, formally described as a signal estimation function $\hat{S} := g(B)$. Several mathematical models $g(B)$ for signal estimation of fMRI or EEG data source exist in present literature (Fransson, Krüger, Merboldt & Frahm, 1999; Glover, 1999). However, as the observed signal is an additive function of neural activity and the multisource noise generated from the device and the experimental setup, the observed signal S can be formulated as:

$$S = f(B) + N \tag{3.27}$$

For encoding brain data, we propose a generalised optimisation based framework based on minimisation of Root Mean Squared Error (RMSE) between an observed signal S and a predicted signal $\hat{S}$. It can be formalised as:

$$\min_{B,\theta} \sqrt{\frac{\sum_t (S - \hat{S}(B,\theta))^2}{t}} \tag{3.28}$$

Such that:

$$B = \mathbb{I}^+ \tag{3.29}$$

$$\sum_t B_t \leq a \tag{3.30}$$

$$0 \leq B \leq 1 \tag{3.31}$$

$$b \leq \theta \leq c \tag{3.32}$$

The above optimiser solves for the RMSE, subject to the following constraints:

1. Binary constraints for spikes: The binary contraint for the spikes are implemented by forcing B to be integer and within range of [0,1].
2. Constraint on the number of spikes: The $\sum_t B_t \leq a$ constraint enforces the maximum number of spikes to be limited to a. This constraint is of major importance from a biological plausibility perspective. Since the encoding scheme discussed here aims to mimic temporal coding behaviour of the human

brain, it is always preferable to encode maximal information with minimal number of spikes.

3. Bounds can be set on the other parameters θ to be optimised as part of the prediction model $S(B, \theta)$

The aforementioned optimisation problem belongs to the paradigm of mixed integer programming, where a subset of parameter or decision variables to be optimised, are integers. In our implementation, we have used a mixed integer Genetic Algorithm (GA) solver (Deep, Singh, Kansal & Mohan, 2009) for solving the above optimisation problem. As opposed to traditional GA solvers that optimise a fitness function, a mixed integer GA minimises a penalty function which include terms for feasibility. This penalty function is combined with binary tournament selection to select individuals for subsequent generations. The reader is directed to Sengupta et al. (2015) for details of this process, and in particular, how *a-priori* knowledge of the data source can be utilised to improve the formalisation.

3.6 Learning & Evolution

In order to take advantage of the powerful spatio-temporal information processing abilities of SNN, we must be able to train them in some manner. By this, we mean that a predictable output sequence must be able to be defined for a given input sequence (or, more commonly, sequence**s**). Without this ability to predict or impose an output sequence on a SNN, their applications are highly limited.

There are a number of complexities in defining learning rules for SNN. The explicit temporal dependence of these types of network results in asynchronous information processing, requiring complex and high-performance software or hardware environments for simulation. In addition, recurrent and self-referential network topologies are often used in SNN. This invalidates the effectiveness of older generation learning methods, such as the form of error backpropagation commonly used in multi-layer perceptrons. Despite these difficulties, a number of different training methods have been proposed. All of these techniques function in a similar manner; through some change in the synapse (generally the alteration of synaptic weights), they simulate an increase or decrease in the amounts of ions passing from the presynaptic neuron to the postsynaptic neuron.

Learning rules for SNN can be categorised into three general paradigms; supervised, semi-supervised or reinforcement, and unsupervised. In supervised learning, we have some output signal we wish to impose on a system; through some method of

error calculation, we explicitly alter parameters in the model to iteratively obtain that output signal. Supervised learning is typically used where there is a good deal of information about the data context, and the features which contribute to a successful output, *e.g.* the identification of a cancer marker in genetic samples. Supervised learning is discussed in the context of SNN in Section 3.6.2. In semi-supervised or reinforcement learning, we have some 'teacher signal' which provides an independent feedback to the learning system, encouraging it to converge on the desired output signal. Finally, in unsupervised learning, we do not define a specific output, and instead allow the algorithm to identify features of interest and generate some output based on its own representation of the data. Here, we can imagine a clustering task; for example, the grouping of unknown products in a digital marketplace based on features rather than brand or model name. Unsupervised learning is discussed in the context of SNN in Section 3.6.1.

Existing SNN learning techniques include Spike Time Dependent Plasticity (STDP) (already discussed), Spike Dependent Synaptic Plasticity (SDSP) (Brader, Senn & Fusi, 2007; Fusi, 2000), Remote Supervised Method (ReSuMe) (Ponulak, 2005; Ponulak & Kasiński, 2006), Tempotron (Gütig & Sompolinsky, 2006), Chronotron (Florian, 2010), deSNN (Dhoble et al., 2012), and the Spike Pattern Association Neuron (SPAN) of Mohemmed, Schliebs, Kasabov and Matsuda (2011).

Despite their differences, most of these techniques function in a similar manner; through some change in the synapse (generally the alteration of synaptic weights), they simulate an increase or decrease in the amounts of ions passing from the presynaptic neuron to the postsynaptic neuron. This mechanism was proposed by Donald Hebb in 1949, and as a result, is generally termed 'Hebbian Learning'. Hebb postulated that

> "When an axon of cell *A* is near enough to excite cell *B* and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that *A*'s efficiency, as one of the cells firing *B*, is increased" (Hebb, 1949)

The alteration of these weights, or 'synaptic plasticity', leads to the amplification or suppression of the effect of presynaptic neurons, in turn leading to a desired or predictable output sequence. In the case of these computational models of SNN, the connection weight between the presynaptic and postsynaptic neurons is changed, amplifying or suppressing the entire input spike sequence from that presynaptic neuron.

A precise temporal coding paradigm is required in some artificial control systems. Examples are neuroprosthetic systems which aim at producing functionally useful movements of paralysed limbs by exciting muscles or nerves with sequences of short electrical impulses (Popović & Sinkjaer, 2000). Precise relative timing of impulses is critical for generating desired, smooth movement trajectories (Kasiński & Ponulak, 2006a). For this purpose, the learning algorithm known as SPAN is appropriate. Other applications, such as classification of emotional states or identification of particular events in streaming data can instead rely on properties other than defined spike times to represent class or regression values.

3.6.1 Unsupervised Learning

For unsupervised learning in SNN, it has been well established that the timing of pre- and post-synaptic spikes determines potentiation and depression of synaptic weights. A number of models have been proposed to explain the unsupervised learning occuring in the mammalian brain. Among the two most popular (and biologically plausible) are Spike Time Dependent Plasticity (STDP) introduced in (among others) the paper by Bi and Poo (1998); and Spike Dependent Synaptic Plasticity (SDSP), introducted in Fusi (2000) and explored further in Fusi (2003). Only STDP will be explored in this thesis, though SDSP has merit for later exploration as it is a technique occasionally used in neuromorphic hardware systems (*cf.* Chapter 7).

3.6.1.1 Spike-Time Dependent Plasticity

Spike Time Dependent Plasticity (STDP) is a temporally asymmetric form of Hebbian learning induced by temporal correlations between the spike timings of pre- and post-synaptic neurons (Markram, Gerstner & Sjöström, 2012; S. Song, Miller & Abbott, 2000). As with other forms of synaptic plasticity, it is thought to underly learning and memory in the brain, as well as the development and refinement of neuronal circuits during brain development (Allen, Celikel & Feldman, 2003; Bi & Poo, 1998). With STDP, repeated presynaptic spike arrival a few milliseconds before postsynaptic action potentials leads (in many synapse types) to Long-Term Potentiation (LTP) of the synapses, whereas repeated spike arrival after postsynaptic spikes leads to Long-Term Depression (LTD) of the same synapse. The synaptic weight change as a function of the relative timing of pre- and post-synaptic action potentials is called the STDP function or 'learning window', and varies between synapse types. See Figure 3.8 for a visual example of the learning window.

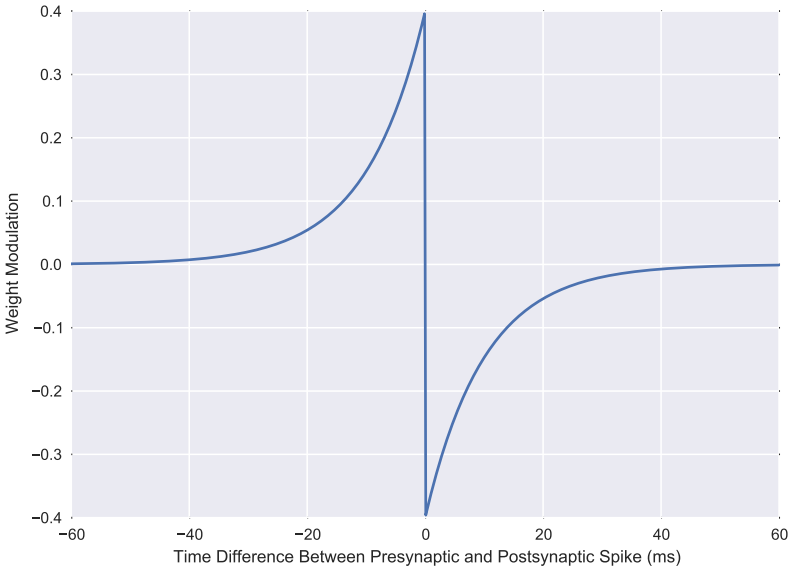


Figure 3.8: An illustration of the STDP learning window, recreated from the canonical formulation in S. Song, Miller and Abbott (2000). The STDP function shows the change of synaptic connections as a function of the relative timing of pre- and post-synaptic spikes. The magnitude of the weight change (y-axis) is modulated by the $A_{+|-}$ parameters and the duration of the weight modulation effect by the respective $\tau_{+|-}$ parameters.

The net result of STDP is that a neuron can discriminate between, and then integrate, temporally significant inputs and transform this into a meaningful output, even though the actual meaning is not strictly known by the neuron (Markram et al., 2012). Networks employing STDP (and indeed, SDSP) operate as palimpsests; that is, older stimuli are forgotten to make room for new ones (Amit & Fusi, 1994; Fusi, 2000).

In general, the synaptic weight is altered based on the timing of post-synaptic action potentials in relation to the pre-synaptic spike. The weight change Δw_j at a synapse from presynaptic neuron j is defined simply as

$$\Delta w_j = \sum_{j=1}^{N} \sum_{n=1}^{N} W\left(t_i^n - t_j^f\right) \tag{3.33}$$

where f and n label the firing times of the pre- and post-synaptic spike times t_j^f of presynaptic neuron j and t^n respectively. Function $W(x)$ denotes an STDP learning window, often defined as

$$W(x) = \begin{cases} A_+ \exp\left(\frac{-x}{\tau_+}\right), & \text{for } x > 0 \\ A_- \exp\left(\frac{-x}{\tau_-}\right), & \text{for } x < 0 \end{cases} \tag{3.34}$$

where A_+ and A_- are generally an empirically determined constant, but can vary with w_j, and time constants τ_+ and τ_- are generally on the order of 10 ms (the "critical window"). The general shape of the learning window is shown in Figure 3.8.

This most common version of STDP eventually (*i.e.* with sufficient learning time) converges in a bimodal distribution of synaptic weights, where each weight assumes its minimal or maximal possible value (Kasiński & Ponulak, 2006a). This observation supports the work in Indiveri, Stefanini and Chicca (2010) on Very Large Scale Integration (VLSI) Application-Specific Integrated Circuit (ASIC) hardware, suggesting that bistable synapses are sufficient for effective learning through synaptic weight updates.

It is noted though, that pre- or post-synaptic dynamics alone are not enough to model this learning properly. Instead, a third factor should be introduced, typically suggested to be a gene expression (Markram et al., 2012; Pawlak, Wickens, Kirkwood & Kerr, 2010). This factor supports the later contention that the CNGM discussed in Section 3.6.3.3 may in the future be an important component for optimal functionality of an SNN-based reservoir that utilises STDP.

STDP has been argued against (Frégnac et al., 2010), but the role of STDP in brain function has been supported in a number of studies including those of Gerstner (2010), Hebb (1949), Konorski (1948), Schulz (2010) and Lisman and Spruston (2010). The effect of STDP is not yet resolved (Bi & Poo, 2001; Kepecs, van Rossum, Song & Tegner, 2002), and it has been postulated that STDP may actually be a byproduct of learning behaviours that optimise other network properties (Chechik, 2003). In any case, there is sufficient evidence that STDP increases the effectiveness of reservoir and SNN computing (Bohte, 2004).

Computationally, STDP updates can be handled as a Lookup Table (LUT) to reduce memory and computational demand. This has been implemented in practice in Pfeil et al. (2012). LUTs do not limit the flexibility of the weight update as long as their weight dependence does not change over time. This LUT can also be projected or mapped to a wider range of weights than the table can contain. Pfeil et al. (2012) contend that 64-bit precision in terms of network accuracy is possible using 8-bit synaptic weights and mapping these values to a wider range of weights. Bounded STDP models such as the one introduced in Gütig, Aharonov, Rotter and Sompolinsky (2003) are the most appropriate to use in such a case. Pfeil et al. implement STDP such that weight updates are accumulated over several steps and applied. This technique

is not common, but it is worth noting as a constraint of this particular hardware based approach.

STDP in hardware is typically implemented for a limited range of synapses (Ramakrishnan, Hasler & Gordon, 2010), and does not generally implement long-term plasticity (Vogelstein, Mallik, Culurciello, Cauwenberghs & Etienne-Cummings, 2007). Software simulations therefore currently allow for greater numbers of plastic synapses and LTD-LTP. New hardware systems (Chapter 7) are beginning to incorporate larger numbers of plastic synapses in an effort to model neuronal behaviours more precisely.

3.6.1.2 Spike Dependent Synaptic Plasticity

Spike Dependent Synaptic Plasticity (SDSP) is a modification of the STDP learning rule, as described in the prior section. In its strictest form, SDSP is an unsupervised method, although a supervised formulation using reference or 'teacher' neurons exists (Brader et al., 2007; Fusi, 2000).

Similar to STDP, presynaptic spikes arriving at the neuron before a postsynaptic spike increases the synaptic weight. At this time, if the postsynaptic membrane potential V_{mem} is greater than some threshold voltage V_ϑ potentiation of the neuron occurs. Similarly, if it is lower than the threshold, depression will occur. Here, this postsynaptic site is represented by its Calcium concentration $C(t)$. The synaptic weight W therefore changes according to

$$W = \begin{cases} W + a, & \text{if } V_{mem} > V_\vartheta \text{ and } \Theta^l_{\text{pot}} < C(t) < \Theta^h_{\text{pot}} \\ W - b, & \text{if } V_{mem} \leq V_\vartheta \text{ and } \Theta^l_{\text{dep}} < C(t) < \Theta^h_{\text{dep}} \end{cases} \tag{3.35}$$

where a and b represent the amount of potentiation and depression respectively; $[\Theta^l_{\text{pot}}, \Theta^h_{\text{pot}}]$ and $[\Theta^l_{\text{dep}}, \Theta^h_{\text{dep}}]$ are the bounds for $C(t)$ for potentiated and depressed states, respectively (Azghadi, Iannella, Al-Sarawi, Indiveri & Abbott, 2014; Fusi, 2000).

In the case that these conditions are not satisfied, there will be no change to the afferent synapses; and in the case that there is no input at all, these synaptic weights will drift into bistability. The direction of this drift will depend on the synaptic weight at the last stimulus, and whether it is above or below a weight threshold Θ_w, as

$$\frac{\mathrm{d}W(t)}{\mathrm{d}t} = \begin{cases} \alpha, & \text{if } W(t) > \Theta_w \\ -\beta, & \text{if } W(t) \leq \Theta_w \end{cases} \tag{3.36}$$

A supervised version of the SDSP rule can be implemented through the use of a reference or 'teacher' neuron, similar to that used in ReSuMe (*cf.* Section 3.6.2.1 or Ponulak, 2005). This teacher neuron is separate from the network except to provide this desired output sequence, which is then used in the calculation of the synaptic weight change. This form of SDSP was used for character recognition, where it showed reasonable levels of effectiveness (Brader et al., 2007).

A significant divergence from STDP in this model is that the synapses are essentially binary on long time scales. Synaptic transitions are initiated only by elevated presynaptic rates. For low presynaptic rates, there are essentially no synaptic transitions. This solves the problem of synaptic stability when using the device in engineering applications (Fusi, 2000). Additionally, this means that these synapses can likely be efficiently implemented by exploiting the bistability of memristors.

3.6.2 Supervised Learning

Supervised learning refers to those techniques which define or impose a specific behaviour as the result of a learning algorithm. Depending on the method of supervised learning, these specifically imposed behaviours could be the generation of a defined spike train from some given input spike trains, or the identification of such a spike train in the context of classification or regression. Here, we introduce the primary established techniques for supervised learning in SNN such as the ReSuMe of Ponulak (2005) and Tempotron introduced in Gütig and Sompolinsky (2006), and emerging techniques such as SPAN (Mohemmed, Schliebs, Kasabov & Matsuda, 2011).

3.6.2.1 Remote Supervised Method

The Remote Supervised Method (ReSuMe) differs from most SNN learning algorithms by incorporating a reference, or 'teacher' neuron to provide the desired output spike sequence for a given neuron (Ponulak, 2005; Ponulak & Kasiński, 2006). It functions by modifying the synaptic weight w of a connection between a presynaptic neuron n^{in} and a postsynaptic neuron n^{out} such that

$$\frac{\mathrm{d}}{\mathrm{d}t}w(t) = \lambda \left[\Phi^{\text{des}}(t) - \Phi^{\text{out}}(t)\right] \left[a + \int_0^\infty W(s)\, \Phi^{\text{in}}(t-s)\, \mathrm{d}s\right] \tag{3.37}$$

where $\Phi^{\text{des}}(t)$, $\Phi^{\text{in}}(t)$ and $\Phi^{\text{out}}(t)$ are the target, pre- and post-synaptic spike trains respectively, where these are defined by the sums of their firing times (Gerstner & Kistler, 2002). The target spike train Φ^{des} is produced by a reference or 'teacher' neuron, separate from the network except to provide this reference signal. The parameter a expresses the amplitude of the non-correlation contribution of the total weight change, while the convolution function defines the Hebbian response of the weight w. The integral kernel $W(s)$ is known as a learning window, which is defined over a time delay s between the spikes occurring in the pre- and post-synaptic neurons. For excitatory synapses a is a positive, and $W(s)$ is of a similar shape to STDP. For inhibitory synapses, this is reversed, with a a negative and $W(s)$ of a similar shape to anti-STDP (Ponulak & Kasiński, 2006). The factor λ is a learning rate adjustment, to increase or decrease the magnitude of the weight modification described by the learning rule. The error between the desired and actual output sequences is generally defined as a cross-correlation of a kernelised version of these.

ReSuMe, when used in conjunction with a Liquid State Machine, is theoretically capable of transforming any input spike train to any output spike train (Kasiński & Ponulak, 2006b). This differentiates it from most other established forms of learning in SNN, as these are typically able to only categorise or classify data, not impose precise spike timing on a sequence. One other such learning method is the SPAN, (*cf.* Section 3.6.2.4).

The Tempotron method (*cf.* Section 3.6.2.2) has been described as a specific, limited application of ReSuMe (Florian, 2008), where $a = 0$, and W replaces ε in Equation 3.38. The abilities of ReSuMe, particularly in that it can impose precise spike timing on an output sequence and discriminate between more than two classes of data, are significantly greater than that of Tempotron for a marginal increase in computational complexity.

3.6.2.2 Tempotron

The Tempotron learning rule (Gütig & Sompolinsky, 2006) functions by modifying the synaptic weights of the input synapses such that the trained neuron emits one spike when presented with inputs corresponding to one category and no spike when the inputs correspond to another category. Before being presented with an input spike train, it is assumed that the neuron's potential is at rest, and that after the neuron emits a spike in response to an input pattern all other incoming spikes are suppressed and have no effect on the neuron. By this, it is meant that even if the neuron would be sufficiently stimulated to fire more than one spike, the spikes following the first one

are artificially suppressed. The membrane voltage u of a trained neuron is modelled as a sum of postsynaptic potentials

$$u(t) = u_0 + \sum_i w_i \sum_{t_i^f < t} \varepsilon\left(t - t_i^f\right) \tag{3.38}$$

where u_0 is the resting potential, w_i is the synaptic efficacy of synapse i, and $\varepsilon\left(t - t_i^f\right)$ describes the form of the postsynaptic potential induced in the neuron i by a spike at t_i^f received from neuron i. The first sum runs over all presynaptic neurons, and the second one runs over all spikes of neuron i prior to t. When u overcomes the firing threshold ϑ, the neuron emits a spike. Tempotron learning minimises the following cost function, for each input pattern:

$$C = \begin{cases} \vartheta - u_{\max}, & \text{if } u_{\max} < \vartheta \text{ and the neuron should fire for this pattern} \\ u_{\max} - \vartheta, & \text{if the neuron fired } (u_{\max} \geq \vartheta) \text{ but should be silent} \\ 0, & \text{otherwise} \end{cases} \tag{3.39}$$

where $u_{\max} = u(t_{\max})$ is the maximal value of the postsynaptic potential u, in the case that the neuron did not fire. In the case that the neuron fired, $u_{\max}$ is the maximal value that u would have been reached if the neuron would have not fired. Applying the gradient descent method in the space of synaptic efficacies for minimising the above cost function leads to the Tempotron learning rule:

$$\Delta w_i = \begin{cases} \lambda \sum_{t_i^f < t_{\max}} \varepsilon\left(t_{\max} - t_i^f\right), & \text{if a desired postsynaptic spike is absent} \\ -\lambda \sum_{t_i^f < t_{\max}} \varepsilon\left(t_{\max} - t_i^f\right), & \text{if an undesired postsynaptic spike is present} \\ 0, & \text{otherwise} \end{cases} \tag{3.40}$$

where $\lambda > 0$ is the learning rate. During learning, synaptic changes Δw_i are applied after each presentation of an input pattern. It can be seen that the dynamics of this learning rule have little biological plausibility, as it requires the monitoring of the maximum of u.

While the rule assumes that the precision of spike times in input patterns is important, it does not apply this same precision to the timing of output spikes. The output pattern is also limited to firing either one or no spikes per input pattern with no level of temporal precision. This behaviour is not supported by the literature, which suggests that a level of precise spike timing in biological neurons is a primary means of information encoding (Bohte, 2004; Nawrot et al., 2009; Stein et al., 2005; Villa et al.,

1999). It is also assumed that learned input patterns are isolated from other inputs and thus that the trained neuron is initially at rest, which is not the case in the brain. This lack of precise spike timing makes it impossible to use the output spikes as a temporally meaningful sequence, limiting the usefulness of the Tempotron method in practice.

3.6.2.3 Chronotron

Similar to ReSuMe, Chronotron is capable of learning spike sequence mappings using the precise timing of spikes. Two forms of the learning rule have been proposed; an analytically derived version termed 'E-learning', and a biologically plausible version known as 'I-learning' (Florian, 2012). E-learning is a gradient-descent optimisation of the synaptic weights to minimise an error function E defined as the difference between the actual output spike train and the desired spike train,

$$E = \sum_{t^f \in \mathcal{F}^*} u\left(t^f\right) - \sum_{\tilde{t}^f \in \tilde{\mathcal{F}}^*} u\left(\tilde{t}^f\right) + \gamma_d \sum_{\substack{(t^f, \tilde{t}^g) \\ t^f \in \mathcal{F} - \mathcal{F}^* \\ \tilde{t}^g \in \tilde{\mathcal{F}} - \mathcal{F}^*}} \sigma\left(\frac{|t^f - \tilde{t}^g|}{\tau_q}\right) \tag{3.41}$$

where $\mathcal{F} = \{t^1, t^2, \ldots, t^n\}$ and $\tilde{\mathcal{F}} = \{\tilde{t}^1, \tilde{t}^2, \ldots, \tilde{t}^n\}$ are the actual and desired spike trains, and $\mathcal{F}^*$ and $\tilde{\mathcal{F}}^*$ are the subsets of these sets representing the spikes that should be removed and added respectively. This difference is measured using a modified version of the Victor and Purpura (VP) distance metric (Victor & Purpura, 1997) that can handle the discontinuities inherent in this measure. The VP metric is one of the two metrics commonly used in neurobiology for quantifying the distance between two spike trains; the other is the Van Rossum metric (van Rossum, 2001). In contrast to the I-learning rule, E-learning implements an off-line learning process that requires the identification of the firing times of all spikes in the network in order to compute the error, similar to the Batch formulation of SPAN. The E-learning rule is defined after derivation with $\sigma(x) = \frac{x^2}{2}$ as

$$\Delta w_j = \gamma \left[\sum_{\tilde{t}^f \in \tilde{\mathcal{F}}^*} \lambda\left(\tilde{t}^f\right) - \sum_{t^f \in \mathcal{F}^*} \lambda\left(t^f\right) + \frac{\gamma_r}{\tau_q^2} \sum_{\substack{(t^f, \tilde{t}^g) \\ \tilde{t}^g \in \tilde{\mathcal{F}} - \mathcal{F}^*}} \left(t^f - \tilde{t}^g\right) \lambda_j\left(t^f\right) \right] \tag{3.42}$$

where γ is a positive learning rate and γ_r is another positive parameter.

The other version of Chronotron, I-learning, is similar to ReSuMe and can be used for online learning in the same manner as Incremental SPAN. In Florian (2010), an extensive analysis was undertaken to demonstrate the performance of the algorithm regarding its learning ability, memory capacity, learning behaviour in the context of noisy input patterns, and the effect of various parameters. The results show that E-learning, although less biologically plausible, achieves better performance in terms of the number of temporal patterns that can be learned compared to both I-learning and ReSuMe. I-learning is defined as

$$\Delta w_j = \gamma \operatorname{sign}(w_j) \left[\sum_{\tilde{t}^f \in \tilde{\mathcal{F}}} I_j \left(\tilde{t}^f \right) - \sum_{t^f \in \mathcal{F}} I_j \left(t^f \right) \right] \tag{3.43}$$

where γ is a positive learning rate and I_j is the synaptic current of a synapse j.

3.6.2.4 Spike Pattern Association Neuron

The Spike Pattern Association Neuron (SPAN) learning algorithm, introduced and extended in Mohemmed and Kasabov (2012), Mohemmed, Schliebs and Kasabov (2011), Mohemmed, Schliebs, Kasabov and Matsuda (2011), Mohemmed, Schliebs, Matsuda, Dhoble and Kasabov (2011), Mohemmed, Schliebs, Matsuda and Kasabov (2013), is primarily an extension of classical error backpropagation using the Widrow-Hoff ('Delta') learning rule. More formally, the Delta rule is defined as

$$\Delta w_i = \lambda \, x_i \, (y_d - y_a) = \lambda \, x_i \, \delta_i \tag{3.44}$$

where $\lambda \in \mathbb{R}$ is a positive learning rate; x_i is the input through synapse i; and y_d and y_a respectively denote the desired and actual neural outputs. Here $\delta_i = y_d - y_a$, or the error between the desired and actual outputs of the neuron.

The difficulty here is applying this rule, developed for use with older generation neurons, to SNNs. In older generation ANNs, the input and output signals of a neuron are real-valued, making the calculation of δ_i trivial. However, in SNNs, we pass spike trains, which require us to seek an alternative approach. In more formal terms, if we consider x_i, y_d and y_a as spike trains $s(t)$ in the form

$$s(t) = \sum_f \delta \left(t - t^f \right) \tag{3.45}$$

where t^f is the firing time of a spike and $\delta(\dots)$ is the Dirac delta function

$$\delta(x) = \begin{cases} 1, & \text{where } x = 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.46}$$

the difference between the actual and desired spike trains does not define an error landscape that can be minimised with gradient descent.

SPAN addresses this through the application of convolution kernels. To effectively define a distance (error) between spike trains, each is convolved with some kernel function $\kappa(t)$. We define

$$\tilde{x}_i(t) = \sum_{t_i^f \in F_{\text{in}}} \kappa\left(t - t_i^f\right) \tag{3.47}$$

$$\tilde{y}_d(t) = \sum_{t_d^g \in F_{\text{d}}} \kappa\left(t - t_d^g\right) \tag{3.48}$$

$$\tilde{y}_a(t) = \sum_{t_a^h \in F_{\text{a}}} \kappa\left(t - t_a^h\right) \tag{3.49}$$

where $F_{\text{in}}, F_{\text{d}}$ and F_{a} are respectively the input, desired and actual spike trains. We then substitute x_i, y_d and y_a with the kernelised spike trains $\tilde{x}_i(t), \tilde{y}_d(t)$ and $\tilde{y}_a(t)$, and obtain a new learning rule for spiking neurons

$$\Delta w_i(t) = \lambda\, \tilde{x}_i(t)\ \left(\tilde{y}_d(t) - \tilde{y}_a(t)\right) \tag{3.50}$$

This equation represents a real-time learning rule, *i.e.* the synaptic weights change over time. By integrating this equation, we derive the batch version of SPAN

$$\Delta w_i = \lambda \int_0^\infty \tilde{x}_i(t)\ \left(\tilde{y}_d(t) - \tilde{y}_a(t)\right)\ \mathrm{d}t \tag{3.51}$$

As noted above, in order to define a suitable error landscape, we convolve the spike trains with a function denoted as $\kappa(t)$. The primary exploration of SPAN has used alpha kernel. This is defined as

$$\kappa_\alpha(t) = e\ \tau^{-1}\ t\ e^{\frac{-t}{\tau}} \Theta(t) \tag{3.52}$$

where $\Theta(t)$ refers to Heaviside function (Equation 3.15) and $\tau \in \mathbb{R}$ is a time constant. We can therefore represent a convolved spike train as

$$\begin{aligned}\tilde{s}(t) &= \sum_{t^f} \kappa\left(t - t^f\right) \\ &= \sum_{t^f} e\,\tau^{-1}\,\left(t - t^f\right)\,e^{-\frac{\left(t-t^f\right)}{\tau}}\,\Theta\left(t - t^f\right)\end{aligned}$$

and using this kernel function, perform the integration of Equation 3.51,

$$\begin{aligned}\Delta w_i &= \lambda \int_0^\infty \Delta w_i(t)\,\mathrm{d}t \\ &= \lambda\left(\frac{e}{2}\right)^2\left[\sum_g \sum_f \left(\left|t_i^f - t_a^h\right| + \tau\right) e^{-\frac{\left|t_i^f - t_d^g\right|}{\tau}} - \sum_h \sum_f \left(\left|t_i^f - t_a^h\right| + \tau\right) e^{-\frac{\left|t_i^f - t_a^h\right|}{\tau}}\right]\end{aligned} \tag{3.53}$$

An interesting study of SPAN was performed in Scott (2012), where the behaviour of this learning rule in the context of memory capacity and network structure effects was explored.

SPAN is a viable candidate for implementation in the NeuCube framework, particularly for the generation of arbitrary spike sequences for motor control, among others.

3.6.3 Evolutionary Methods

Evolutionary Methods are so called because they share the property of 'evolution', in the sense that they are adaptive and dynamic with regards to their structure and behaviour over time. A canonical example of this is the Evolving Connectionist System (eCoS) family of methods discussed in Kasabov (2015), and the current state of the art deSNN (Dhoble et al., 2012) introduced here. The interested reader is directed to the reviews of this area in Schliebs and Kasabov (2013) or Watts (2009).

3.6.3.1 Evolving Spiking Neural Network

Based on the eCoS principles, an Evolving Spiking Neural Network (eSNN) architecture was proposed in Kasabov (2007) and Wysoski, Benuskova and Kasabov (2010). It was initially designed as a visual pattern recognition system. The first eSNNs were based on the Thorpe's neural model (Thorpe, 2001), in which the importance of early

spikes (after the onset of a certain stimulus) is boosted, called rank-order coding and learning. Synaptic plasticity is employed by a fast supervised one-pass learning algorithm.

These eCoS principles were established in Kasabov (1998):

1. Fast learning from large amount of data, *e.g.* using 'one-pass' training, starting with little prior knowledge;
2. Adaptation in a real time and in an on-line mode where new data is accommodated as it comes based on local learning;
3. 'Open', evolving structure, where new input variables (relevant to the task), new outputs (*e.g.* classes), new connections and neurons are added/evolved 'on the fly';
4. Both data learning and knowledge representation is facilitated in a comprehensive and flexible way, *e.g.* supervised learning, unsupervised learning, evolving clustering, 'sleep' learning, forgetting/pruning, fuzzy rule insertion and extraction;
5. Active interaction with other eCoSs and with the environment in a multi-modal fashion;
6. Representing both space and time in their different scales, *e.g.*: clusters of data, short- and long-term memory, age of data, forgetting, *etc.*;
7. System's self-evaluation in terms of behaviour, global error and success and related knowledge representation.

The main advantage of the eSNN when compared with other supervised or unsupervised SNN models is that it is computationally inexpensive and boosts the importance of the order in which input spikes arrive, thus making the eSNN suitable for on-line learning with a range of applications. For a comprehensive review of eSNN see Wysoski et al. (2010).

An eSNN evolves its structure and functionality in an on-line manner, from incoming information. For every new input data vector, a new output neuron is dynamically allocated and connected to the input neurons. The neuron's connections are initially established using the Rank-Order Learning (RO) rule for the output neuron to recognise this vector (frame, static pattern) or a similar one as a positive example. The weight vectors of the output neurons represent centres of clusters in the problem space and can be represented as fuzzy rules (Soltic & Kasabov, 2010).

In some implementations neurons with similar weight vectors are merged based on the Euclidean distance between them. That makes it possible to achieve a very fast

learning (only one pass may be sufficient), in both supervised and unsupervised modes (Dhoble et al., 2012). When in an unsupervised mode, the evolved neurons represent a learned pattern (or a prototype of patterns). The neurons can be labelled and grouped according to their class membership if the model performs a classification task in a supervised mode of learning. One specific architecture of the eSNN paradigm is explained here.

3.6.3.2 Dynamic Evolving Spiking Neural Network

The Dynamic Evolving Spiking Neural Network Classifier (deSNN) introduced in Dhoble et al. (2012) and more comprehensively in Kasabov, Dhoble, Nuntalid and Indiveri (2013), combines the rank-order learning of Thorpe (*cf.* Section 3.3.4) and temporal learning such as the STDP or SDSP algorithms (*cf.* Section 3.6.1.1 & 3.6.1.2). The initial values of synaptic weights are set according to the rank-order learning principle, which assumes the first incoming spikes are more important than the rest. The weights are further modified to accommodate following spikes activated by the same stimulus, with the use of a temporal learning rule.

Rank-order learning has several advantages when used in SNN, mainly: fast, one-pass learning (as it uses the extra information of the order of the incoming spikes) and asynchronous data entry (synaptic inputs are accumulated into the neuronal membrane potential in an asynchronous way). The postsynaptic potential of a neuron i at a time t is calculated as:

$$\mathrm{PSP}(i,t) = \sum \rho^{\mathrm{order}(j)}\, W_{i,j} \tag{3.54}$$

Where ρ is a modulation factor; j is the index for the incoming spike at synapse i, j and $w_{i,j}$ is the corresponding synaptic weight; order(j) represents the order (the rank) of the spike at the synapse ji among all spikes arriving from all m synapses to the neuron i. The order(j) has a value 0 for the first spike and increases according to the input spike order. An output spike is generated by neuron i if the PSP(i, t) becomes higher than a threshold $\mathrm{PSP}_{\vartheta}(i)$.

During the training process, for each training input pattern (sample, example) a new output neuron is created and the connection weights are calculated based on the order of the incoming spikes. In the eSNN, the connection weights of on-line created connections between a neuron n_i, representing an input pattern of a known class, and an activated input (feature) neuron n_j, are established using the RO rule of

Thorpe and Gautrais (1998):

$$\Delta W_{i,j} = \rho^{\text{order}(i,j(t))} \tag{3.55}$$

After the whole input pattern (example) is presented, the threshold of the neuron n_i is defined to make this neuron spike when this or a similar spatio-temporal pattern (example) is presented again in the recall mode. The threshold is calculated as a fraction (C) of the total PSP, as:

$$\text{PSP}_{\text{max}} = \sum_{j=1}^{m} \sum_{t=1}^{T} \left(\rho^{\text{order}(i,j(t))} W_{i,j(t)} \right) \tag{3.56}$$

and

$$\text{PSP}_{\vartheta} = C \times PSP_{\text{max}} \tag{3.57}$$

If the connection weight vector of the trained neuron is similar to the one of an already trained neuron in a repository of output neurons for the same class, the new neuron will merge with the most similar one, averaging the connection weights and the threshold of the two neurons (Kasabov, 2007; Wysoski et al., 2010). Otherwise, the new neuron will be added to the class repository. The similarity between the newly created neuron and a training neuron is computed as the inverse of the Euclidean distance between weight matrices of the two neurons.

In a typical eSNN, these weights are calculated once and do not evolve further. Dhoble et al. (2012) contend that this is appropriate for static pattern recognition, but not for signals which may change over time. In the latter case, the weights need to be adjusted based on the subsequent spikes, using some form of temporal learning (*i.e.* STDP or SDSP). While the RO learning will set the initial values of the connection weights for a spatio-temporal pattern recognition system utilising the existing event order information, the temporal learning process will adjust these connections based on following spikes as part of the same spatio-temporal pattern. In this way, we extend an eSNN to a deSNN, by incorporating this *dynamic* behaviour. See Listing 3.3 for a pseudocode implementation of this straightforward algorithm.

```
SET deSNN parameters (ρ, C, Sim, SDSP)

for (input i represented as AER):
  create a new output neuron j
  calculate values of connection weights using RO learning rule
  adjust connection weights w_j for consecutive spikes with SDSP
  calculate PSP
```

```
  calculate threshold value
  if(w_j is similar to existing w_j)
    merge neurons
  end if
end for
```

Listing 3.3: Pseudocode implementation of the deSNN Learning Algorithm.

3.6.3.3 Neuro-Genetic Regulatory Network

Properties of all biological cell types, including neurons, are determined by the proteins expressed within them. In turn, these proteins are determined by gene expressions in relation to some internal or external stimulus (Lodish et al., 2012). The properties of these cells (in this case, the neurons) then determine the structural and functional dynamics of the network they exist within, leading to gene expressions determining the behaviour of a neural network (Benuskova & Kasabov, 2008). Therefore, when modelling biological behaviours through SNN, it can be extremely important for us to take these gene expressions into account. This leads us to the CNGM introduced and extended in Benuskova and Kasabov (2008), Kasabov, Benuskova and Wysoski (2005a, 2005b, 2005c, 2011), Marnellos and Schreiber (2003), Mjolsness, Sharp and Reinitz (1991), Storjohann and Marcus (2005), Watts and Kasabov (1999) and Kasabov (2012a).

In this approach, gene expressions control neural parameters, which are no longer constant over a simulation. Through optimising these gene interactions, the initial gene and protein expressions and neural parameters, an optimal state for the neural network can be evolved (Benuskova & Kasabov, 2008). This is typically known as a Gene Regulatory Network (GRN).

A general computational neurogenetic model, as described in the previously introduced papers, but particularly from Kasabov et al. (2005a, 2005b, 2005c) and Benuskova and Kasabov (2008), is presented below.

In general, we consider two sets of genes – a set $\mathbf{G}_{\text{gen}}$ that relates to general cell functions and a set $\mathbf{G}_{\text{spec}}$ which defines specific neuronal information processing functions (*e.g.* receptors, ion channels, etc.) The two sets together form a set $\mathbf{G} = \{G_1, G_2, \ldots, G_n\}$. We assume that the expression level of each gene $g_j(t + \Delta t)$ is a nonlinear function of the expression levels of all the genes in $\mathbf{G}(t)$ as

$$g_j(t + \Delta t) = \sigma\left(\sum_{k=1}^{n} w_{jk} g_k(t)\right) \tag{3.58}$$

We work with normalised gene expression values in the interval $(0, 1)$. The coefficients $w_{ij} \in (0, 4)$ are elements of the square matrix $\mathbf{W}$ of gene interaction weights. Initial values of gene expressions are small random values, generally $g_i(0) \in (0, 0.1)$.

In the current model, we assume a simple scenario where one protein is coded by one gene; the relationship between the protein level and the gene expression level is linear; and protein levels lie between the minimal and maximal values. The protein level $p_j(t + \Delta t)$ is therefore expressed by

$$p_j(t + \Delta t) = \left(p_j^{\max} - p_j^{\min}\right) \sigma\left(\sum_{k=1}^{n} w_{jk} g_k(t)\right) + p_j^{\min} \tag{3.59}$$

The delay Δt corresponds to time intervals of gathering the expression data for genes and proteins. The GRN model exposed above can be integrated with an SNN model into a CNGM. To reduce model complexity and make the large number of parameters manageable, a few simplifications are usually made. Specifically that:

1. Each neuron has the same gene regulatory network
2. Each GRN starts from the same values of gene expressions
3. There is no feedback from neuronal activity or other external factors to the GRN
4. Delays Δt are the same for all genes/proteins.

Some proteins are directly related to neuronal parameters p_j, such that $P_j(t) = P_j(0)p_j(t)$ where $P_j(0)$ is the initial value of the neuron parameter at time $t = 0$. In this way, the gene/protein dynamic can be linked to the dynamics of a neural network. For the purposes of this thesis, a number of genes are potentially useful. However, to simplify this introduction, four specific genes will be discussed here. The effect of these four gene-protein pairings (Table 3.2) is modelled in the following equation, which utilises them to represent four distinct synaptic types and their respective contributions to a generic neuron's membrane voltage or Post-Synaptic Potential (PSP). Other genes (*e.g.* mGLuR3, Jerky, GLAR1) could be introduced to the model later, to model the effects of different genetic neurodegerative illnesses.

This information is used to calculate the contribution of each of the different synapses to a neuron's postsynaptic potential, as

$$\varepsilon_{ij}(s) = A\left(e\left(-\frac{s}{\tau_{ij}^{-}}\right) - e\left(\frac{s}{\tau_{ij}^{+}}\right)\right) \tag{3.60}$$

Protein Expression	Synaptic Effect
AMPA	Fast Excitation
NMDA	Slow Excitation
$GABA_A$	Fast Inhibition
$GABA_B$	Slow Inhibition

Table 3.2: Basic proteins and their synaptic effects, to be utilised in the GRN

where τ_{ij}^{+} and τ_{ij}^{-} are time constants represeting the rise and fall of an individual PSP; and A is the PSP's amplitude; ε_{ij} represents the activity on the synapse between neurons n_i and n_j, modelled separately for fast excitation or inhibition, and slow excitation and inhibition (Kasabov, 2012a; Kasabov et al., 2011).

3.6.3.4 Quantum-Inspired Optimisation

QeSNNs use the principle of superposition of states to represent and optimize features (input variables) and gene parameters of an eSNN model (Kasabov, 2007). They are optimized through quantum inspired genetic algorithm (Defoin-Platel, Schliebs & Kasabov, 2009) or QiPSO. Features or genes are represented as qu-bits in a superposition of 1 (selected), with a probability α, and 0 (not selected) with a probability β. When the model has to be calculated, the quantum bits 'collapse' into a state of either 1 or 0. QeSNN need to be developed further in terms of both theory and applications. The interested reader is directed to the PhD thesis of Schliebs (2010), which covers this topic in depth.

3.7 Reservoir Computing

Reservoir computing is conceptually simple. We define a 'reservoir' as a recurrently connected network of computational elements, that has a fixed structure. In our case, these computational elements are spiking neurons, but in the case of the Echo State Network (ESN) of Jaeger (2001) these can be traditional real-valued artifical neurons. In this way, they are a specific case of a RNN (Goudarzi, Banda, Lakin, Teuscher & Stefanovic, 2014). Maass, Natschläger and Markram (2002) demonstrated that a reservoir computing method had the capacity to approximate any time series, although this contention has been challenged (Hazan & Manevitz, 2012).

The internal structure of this reservoir is fixed, and does not change over the life cycle of the network, including when the network is 'trained' (Lukoševičius & Jaeger,

2009). In effect, this reservoir acts to raise the input data to a higher dimensional state, in much the same way as the SVM kernel machine. This property is discussed in Section 4.2. Their computational capacity is theoretically attributed to a short-term memory property exhibited by reservoirs, which retains and separates temporal sequences from input signals (Jaeger, 2007).

A simple readout mechanism (usually a traditional linear artifical neuron) is then trained on the spiking states of the reservoir. This readout function is the only component of the reservoir which changes after it is initially created (Verstraeten, Schrauwen, D'Haene & Stroobandt, 2007). A benefit of this computational paradigm is in terms of training cost over a more dynamic RNN, as the complex reservoir does not change its structure over time. This mechanism takes advantage of the property demonstrated in Schiller and Steil (2005), where it was shown that in traditional RNN learning methods where all network weights can adapt, the only significant weight changes occur in the output layer.

Here, we define the Liquid State Machine (LSM), the primary form of reservoir computing when dealing with SNN. For more information on reservoir computing, the reader is directed to the review of Lukoševičius and Jaeger (2009), Maass, Markram and Natschläger (2002), and Maass and Markram (2004).

3.7.1 Echo State Networks

The Echo State Network (ESN) of Jaeger (2001) is an attempt to exploit the theoretical benefits of RNN systems while simultaneously avoiding issues implicit in them such as the vanishing gradient problem. A large reservoir of sparsely connected (*i.e.* $< 1\%$ connections in a network) non-spiking neurons are initially generated. The connectome and weighting of connections is generated randomly, and are fixed at the time of generation. Similarly, input neurons are chosen at random, and their input weighting generated the same way. No learning or evolution occurs inside the reservoir. The intent is that the sparse recurrent connections allow a short-term memory of a signal due to its propagation around the network. This memory is colloquially termed an 'echo' (Jaeger, 2002).

Learning in an ESN is limited to the output layer. Here, some mechanism – typically simple error backpropagation – is used to adapt the weighting of the connections between the reservoir and the output neurons (Lukoševičius, 2012). Learning is therefore a fairly efficient process, as only a small number of weights are adapted during the process.

Readers interested in a more comprehensive review of the ESN are directed to Jaeger (2007).

3.7.2 Liquid State Machine

The Liquid State Machine (LSM) introduced in Maass, Natschläger and Markram (2002) is a specific form of reservoir computing (Verstraeten et al., 2007) that constructs a random RNN of spiking neurons. All parameters of the network (*i.e.* synaptic weights, connectivity, delays, and neural parameters) are randomly chosen and fixed during simulation. Such a network is also referred to as a 'liquid'. This liquid serves as an unbiased analog (fading) memory about current and preceding inputs to the circuit. If excited by an input stimulus, the liquid exhibits very complex non-linear dynamics that are expected to reflect the inherent information of the presented stimulus.

The response of the network can be interpreted by a learning algorithm. Figure 3.9 illustrates the principle of the LSM approach. As a first step in the general implementation of the LSM a suitable liquid is chosen. This step determines for example, the employed neural model along with its parameter configuration, as well as the connectivity strategy of the neurons, network size and other network-related parameters.

After creating the liquid, so-called 'liquid states' $x(t)$ can be recorded at various time points in response to numerous different (training) inputs $u(t)$. Finally, a supervised learning algorithm is applied to a set of training examples of the form $(x(t), v(t))$ to train a readout function f, such that the actual outputs $f(x(t))$ are close to $v(t)$. Maass, Markram and Natschläger (2002) and Maass and Markram (2004) argue that the LSM has universal computational power – *i.e.* that it can potentially act as a Turing machine. As previously mentioned, a very appealing feature of the applied training method (the readout function) is its simplicity, since only a single layer of weights is actually modified. A linear training method is therefore sufficient. This is of course in contrast to the forms of 'deep learning' introduced in Hinton, Osindero and Teh (2006), LeCun, Kavukcuoglu and Farabet (2010) and Bengio and LeCun (2007), which require a considerable amount of training time and computational cost.

Hazan and Manevitz (2012) argue that, in contrast to the contention in Maass, Natschläger and Markram (2002), the LSM alone is not sufficient to model human brain functionality. To resolve this issue it is necessary to incorporate some topological constraints – most particularly, the 'small world' connectivity property – on the

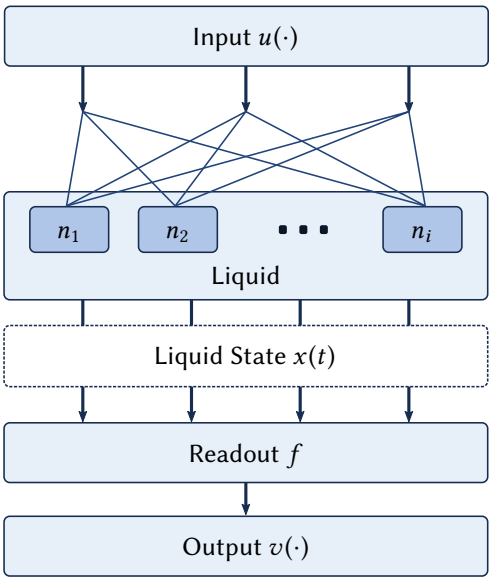


Figure 3.9: Block diagram of the basic principle of a Liquid State Machine (LSM). The liquid transforms inputs u to neurons n into a 'liquid state' x which is in turn mapped by a readout function f into the output v of the network. Figure recreated from Maass, Markram and Natschläger (2002).

network. In addition, the effectiveness of the network is heavily dependent on the random selection of parameters; while such systems are generally robust to noise, their structure and behaviour is not informed by the application. It is further argued that there is no way to extract knowledge from the network, *i.e.* it is a black box. While some work has been done to rectify this last issue – for example, the work of Ozturk, Xu and Príncipe (2007) – this is still a difficult task, and one not yet solved.

These contentions are resolved in the context of the NeuCube framework in Section 4.2, as the NeuCube's structured reservoir shares computational properties with this basic paradigm.

3.8 Simulation Platforms

At some point, we must actually implement these models and algorithms in some system that simulates their dynamics efficiently and accurately. To resolve this, a number of simulation platforms have been created. These simulation platforms provide computational units representing neurons and synapses, which can be interacted with in some way to create our desired network. We can broadly categorise them into two main types of simulation system; software simulators, and neuromorphic hardware systems.

3.8.1 Software Simulators

Software simulation systems are typically a more flexible approach to SNN dynamics modelling. By flexible, it is meant that it is possible to implement a large number of neuronal models on the same computational platform, without the need to physically change the device. Typically, this flexibility comes at a performance cost; software simulation typically requires high-performance computing platforms such as clusters or supercomputers, which are expensive in terms of both power requirements and monetary cost. In this section, we briefly introduce the three most common software simulators for engineering purposes. Other simulators, such as GENESIS and CarlSIM exist, but are typically directed more towards computational neuroscience applications rather than the domain in which we intend to utilise them.

Brian

Brian is an open source Python package for developing simulations of networks of spiking neurons. Introduced in Goodman and Brette (2008), Brian is aimed at researchers developing models based on networks of spiking neurons. Goodman and Brette claim that it is intended to minimise development time, with execution speed a secondary goal. Brian is unique in that users specify neuron models by giving their differential equations in standard mathematical form (plain text), which is interpreted by Brian itself. As a result of this, Brian is capable of modelling any neural behaviour which can be represented in this manner. This feature is intended to make the process as flexible as possible, so that researchers are not restricted to using a limited number of neuron models typically included in simulators. It is written in Python, using the NumPy and SciPy numerical and scientific computing packages. Parts of the simulator can optionally be run using C code generated procedurally. Computationally, Brian uses vectorisation techniques, so that for large numbers of neurons, execution speed is of the same order of magnitude as C code – claimed to be around 25% slower on the Brian website[2], although this claim is unverified in any peer-reviewed sources.

PyNEST

The Neural Simulation Tool NEST is a computer program for simulating large heterogeneous networks of point neurons or neurons with a small number of compartments. NEST is best suited for models that focus on the dynamics, size, and structure of

[2] http://www.briansimulator.org

neural systems rather than on the detailed morphological and biophysical properties of individual neurons (Gewaltig & Diesmann, 2007). It was officially introduced in Diesmann and Gewaltig (2001), but versions of this environment were in varying stages of development and use as early as 1995. NEST's original inteface language is known as SLI, but more commonly in recent developments, a Python interface called PyNEST is used. Extensions to neural behaviour can be implemented in C++, requiring recompilation of the whole software package. In order to increase accuracy over that of other simulation environments, the size of the integration step h can be chosen independent of the simulation step Δt. Models with approximate differential equation solvers such as Runge-Kutta can set the h an order of magnitude smaller than Δt. Within a simulation step Δt it is also possible to solve the equations with an adaptive step-size, which is unique to NEST.

NEURON

Formally introduced in Hines and Carnevale (2001), Neuron is a simulation environment for modelling individual neurons and networks of neurons. This simulator is primarily aimed at computational neuroscience applications, as Neuron models individual neurons at a microscopic level, simulating the transfer of ions through membrane channels in as precise a manner as possible. It is, therefore, as biologically plausible as current microscopic models of neural behaviour.

The primary scripting language that is used to interact with it is 'hoc', but a Python wrapper for the library is also available. Simulations can be written interactively in a shell, or loaded from file. Neuron supports local parallelisation, and distributed computing over clusters via Message-Passing Interface (MPI). The properties of the membrane channels of the neuron are simulated using compiled mechanisms written using the NMODL language or by compiled routines operating on internal data structures that are set up with a GUI tool ('ChannelBuilder'). Interestingly, networks instantiated the GUI tool actually execute faster than identical mechanisms specified with NMODL. Their states and total conductance can be simulated as deterministic, or stochastic. Neuron's ability to model biophysically realistic network dynamics is vital in computational neuroscience applications, but will introduce unnecessary overhead for our more abstracted engineering applications.

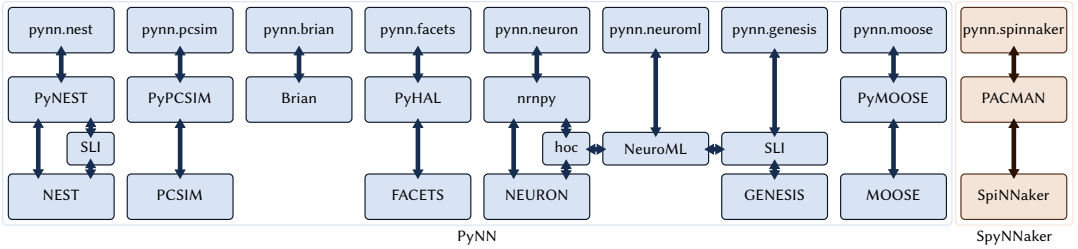


Figure 3.10: A map of PyNN and sPyNNaker simulator interface coverage. Blue (leftmost) boxes represent simulators supported in standard PyNN. The orange (rightmost) box represents sPyNNaker coverage, the SpiNNaker device. The topmost row of boxes shows the PyNN package used to interface with the simulator shown in the bottom boxes. The middle boxes represent that simulator's traditional interface language. Figure recreated from Davison et al. (2008) with the addition of sPyNNaker.

3.8.2 Neuromorphic Hardware Simulators

Neuromorphic hardware systems provide a power-efficient computation platform for the simulation of SNN. We will not discuss these devices in any specifics here; see Chapter 7 for a detailed discussion of available neuromorphic hardware systems and the considerations for implementing the NeuCube framework on the same.

3.8.3 PyNN

PyNN[3] is a simulator interface library for building neuronal network models, where a user can write a single model and have it run on any of the simulators PyNN supports. First introduced in Davison et al. (2008), PyNN is quickly becoming a *de facto* language for neuromorphic hardware systems. See Figure 3.10 for a visual overview of the PyNN's simulator support. As discussed later in Section 7.4, Galluppi, Rast, Davies and Furber (2010) have introduced a version of PyNN known as sPyNNaker, for use on the SpiNNaker neuromorphic hardware device.

The primary advantage of PyNN is that a single simulation can be run on multiple simulation platforms. In this way, we can create one simulator-agnostic version of the NeuCube which can be implemented in a vast number of environments; in conjuction with the small changes needed for sPyNNaker, and a version written in PyNCS (Section 7.3.2), a fairly comprehensive coverage of software and hardware simulation platforms can be supported with only minor code changes.

[3] http://neuralensemble.org/PyNN

3.9 The NeuCube: A New Spiking Neural Network Framework

The NeuCube exploits the principle that structural and functional links within the data are significant and meaningful. It explicitly incorporates temporal, spatial, and spectral dynamics (Kasabov, 2012b; Kasabov et al., 2015; Scott et al., 2013) to represent complex data in a manner similar to the way our brains would process it. This system, in a very rough form and without significant aspects such as the neurogenetic regulatory system, has already proven itself in J. Hu et al. (2014), Kasabov, Hu et al. (2013), D. Taylor et al. (2014) and others to be a state of the art tool for the classification of SSTD. Further work on this system should improve its performance as a general SSTD classification tool, and its applicability to the identified need for a tool like it in BCI, neurorehabilitation, and clinical tasks. The following chapters will introduce this framework, guidelines for its design and implementation, and concrete versions of it on both commodity computers and specialised hardware. Case studies in the fields of neuroinformatics and radioastronomy will also be introduced.

Contributions of this Chapter

1. A review of the existing state-of-the-art in Spiking Neural Networks, including:
   (a) Their biological motivation;
   (b) Principles of spike coding;
   (c) Methods of encoding data into spikes;
   (d) Common and emerging models of spiking neurons;
   (e) Learning and evolution;
   (f) Contemporary architectures for networks of SNN; and
   (g) Simulation tools and technologies

Peer Reviewed Literature With Direct Contributions from this Chapter

1. Sengupta, N., **Scott, N. M.**, and Kasabov, N. (2015). Framework For Knowledge Driven Data Encoding For Brain Data Modelling Using Spiking Neural Network Architecture. In *Proceedings of the 5th International Conference on Fuzzy and Neural Computing*. 17–19 December 2015. Hyderabad, India. Springer. doi:10.1007/978-3-319-27212-2_9
2. Kasabov, N., **Scott, N. M.**, Tu, E., Marks, S., Sengupta, N., Capecci, E., Othman, M., Doborjeh, M., Murli, N., Hartono, R., Espinosa-Ramos, J.I., Zhou, L., Alvi, F., Wang, G., Taylor, D., Feigin, V., Gulyaev, S., Mahmoud, M., Hou, Z.-G. and Yang, J. (2016). Evolving Spatio-Temporal Data Machines Based on the NeuCube Neuromorphic Framework: Design Methodology and Selected Applications. *Neural Networks*. Special Issue on Learning in Big Data. Elsevier. doi:10.1016/j.neunet.2015.09.011

Chapter 4

The NeuCube Framework

> Swiftly the brain becomes an enchanted loom, where millions of flashing shuttles weave a dissolving pattern –
> always a meaningful pattern –
> never an abiding one.
>
> — Charles Sherrington
> (*Man on his Nature*, 1940)

The NeuCube, a form of the RNN reservoir computing paradigm, and an extension of the LSM, is a novel SNN framework for the classification and analysis of Spatio- and Spectro-Temporal Data (SSTD). It explicitly incorporates *a-priori* information about the nature and source of data, and is capable of learning and evolving on-line. In this chapter, this framework and its place in the literature is introduced. This chapter is primarily a review of the NeuCube in the existing literature; it has been separated from the previous review section in order to discuss it in more depth.

The basic framework of the NeuCube was established in a theoretical sense in Kasabov (2012b) and more comprehensively in Kasabov (2014). It was initially codified into a concrete software system in Scott et al. (2013), and has subsequently been extended in terms of both applications and theory in a number of papers. For a review of the development of the NeuCube outside the context of this thesis, the reader is directed to Kasabov et al. (2015), which covers the history of the NeuCube to this point and the various applications it shows promise in.

The intention of the NeuCube was originally to support the creation of modular integrated systems, where those modules corresponded to functions of the brain, and the system as a whole could be integrated together for brain signal pattern

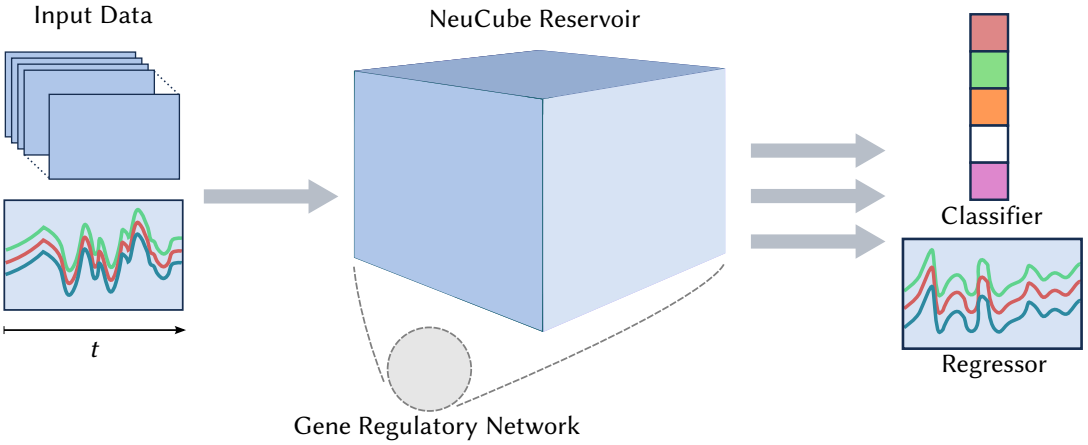


FIGURE 4.1: Block diagram of the NeuCube framework. From left to right, we note the: input data encoding component; feeding into the NeuCube 3-D reservoir component; the optional gene regulatory network that modulates the behaviour of the the reservoir; and the output or classification component.

recognition (Kasabov, 2012b). The framework was originally introduced for the explicit purpose of brain data classification; it is only subsequent to that original paper that it has been extended for use on general SSTD. In straighforward terms, the following components[1] comprise a standard NeuCube model:

1. Input Encoding (Section 4.1);
2. NeuCube 3D Reservoir (Section 4.2); and,
3. Output and Classification (Section 4.3)

With the optional components:

4. Gene Regulatory Network (Section 4.4); and,
5. Visualisation (Section 4.5).

These components are discussed in their respective sections, and a basic block diagram of the framework is shown in Figure 4.1.

The NeuCube is consistent with the contention in Lemm et al. (2011, pp 397) that "...a close interdisciplinary interaction between paradigm and computational model [is] essential." It is on this conceptual foundation that the NeuCube is built, explicitly incorporating spatial, spectral, temporal, and potentially genetic dynamics (Kasabov, 2012b; Scott et al., 2013).

[1]The original architectural nomenclature established in (Kasabov, 2012b) defined 'Components' as 'Modules'. It has been changed in this thesis to avoid confusion with the software 'modules' introduced in Section 6.2

According to Kasabov (2014), the process of creating a NeuCube model for a given SSTD is performed in the following steps:

1. Encode the data into spike sequences: continuous value input information is encoded into trains of spikes;
2. Construct and train in an unsupervised mode a recurrent 3D SNN reservoir to learn the spike sequences that represent individual input patterns;
3. Construct and train in a supervised mode an evolving SNN classifier to learn to classify different dynamic patterns of the reservoir activities that represent different input patterns from SSTD belonging to different classes;
4. Optimise the model through iteration of steps 1–3 above for different parameter values until maximum accuracy is achieved.
5. Recall the model on new data.

Each of these primary components will be briefly introduced in the following sections. In this chapter, specifics of these components with regard to a practical implementation will not be discussed; these discussions will be introduced later in this thesis. Chapter 5 will go into depth regarding our informed choices of discrete components (including encoding, reservoir structure, and classifier) and how these decisions may affect the behaviour of the NeuCube as a whole. Chapter 6 introduces a practical software implementation of this framework, and design considerations which will be generally applicable to further versions of the NeuCube. Neuromorphic hardware implementations are introduced in Chapter 7, and further considerations for these specific computational substrates are discussed there. Therefore, this chapter will be brief; here, it is merely intended to introduce the general framework and conceptual overview of the NeuCube. Primary contributions are discussed later in this thesis.

4.1 Input Encoding

This component is responsible for the conversion of input data to trains of spikes, following the encoding principles established in Section 3.5. Specific methods of input encoding, including the deSNN and BSA techniques will not be discussed here, and can be found in their respective sections (Section 3.5.1–3.5.4) in Chapter 3. Considerations for their use in the NeuCube with respect to algorithm and parameter choice are discussed in Section 5.1.

4.2 NeuCube Reservoir

The NeuCube reservoir is arguably the most complex and novel component of this framework. A major contribution of this thesis is the formulation of design rules for these reservoirs. Specifics of reservoir design are introduced in Section 5.2.

The reservoir in the NeuCube architecture can be considered to be an extension of the LSM discussed in Section 3.7.2. It shares a number of conceptual similarities to the LSM, primary among which is the fading memory and ability to transform otherwise non-linearly separable data to a higher dimensional state. Recall from that section the limitations of that architecture, particularly the issues raised by Hazan and Manevitz (2012):

1. In contrast to the contention in Maass, Natschläger and Markram (2002), the LSM alone is not sufficient to model human brain functionality.
2. The effectiveness of the network is heavily dependent on the random selection of parameters.
3. There is no way to extract knowledge from the network.

The NeuCube reservoir seeks to address these issues through meaningful structure of the network. The random connectome of the LSM is replaced with a meaningful, small-world neuronal structure and connectome. This network structure and connectome is designed to physically encode our *a-priori* knowledge of the data we are processing. This allows us to inspect the evolution of areas of the model based on its unsupervised learning, and extract some meaning from this. In contrast, with the random structure and connectome of the LSM, no meaningful insights can be drawn from the way that the connectome evolves. This is related to the fact that a canonical LSM has a fixed connectome and performs no internal learning or modification of its synapses. This feature ensures that in conjunction with common neural network analysis techniques to extract knowledge from the structure of the network, we can also visualise it in a meaningful way. This means that general patterns, abberant behaviours, and insights not otherwise able to be identified with traditional signal processing can be surfaced to the researcher. Additionally, this represents a similar processing paradigm to the brain; the structure is meaningful, and not arbitrarily generated. More complex models can incorporate heterogenous neuronal or synaptic populations in different areas of the model, in some way providing the ability to represent the biological properties found in these areas. In this way, we primarily address issue three of the above list, with some benefit to issues one and two.

Unlike the LSM only some network parameters are chosen at random; most particularly, the initial starting weights of the synapses. In the case of the NeuCube, these are no longer fixed, and can evolve with the incoming data using the STDP learning rule. By incorporating these key features, the NeuCube framework can resolve the issues raised, as the random initialisation of the network is no longer its final state. It can, instead, evolve and respond to the input data, and learn over time. Additionally, if the network is intially generated in a suboptimal state, the system can recitfy this autonomously, minimising any impact of poor starting state. This directly addresses issue two of the above list.

The reservoir structure was defined abstractly in Kasabov (2012b) to spatially map brain areas for which data is available. Scott et al. (2013) introduced a concrete mapping for reservoir structures in a neuroinformatics context, based on the Talairach Atlas. This mapping is discussed further in the context of reservoir design in Section A.1, and applied to a neuroinformatics data classification task in Section A.2. General SSTD reservoir designs are discussed in Section 5.2, where some guidelines for the topology of NeuCube reservoirs for arbitrary data are introduced. Here, we can address the first issue of the above list through the intelligent construction of a NeuCube reservoir structure. The LSM cannot sufficently model brain functionality due to its lack of capacity for evolution, and the fact that it is an arbitrary structure. Through the use of a meaningfully shaped reservoir based on *a-priori* knowledge of the structure of a general human brain, we are able to create a model which directly encodes the spatial information in a neuroinformatics data set into the reservoir. Further sophistication could be added to this model if more information was known. For example, we could generate the connectome based on information from DTI, which provides a measure of density and directionality of synaptic tracts in the brain. Neuron populations in the model could be based on knowledge of the biological properties of neurons in given areas of the subject brain, further encoding context into the model. The model's evolution over time can therefore be interpreted in context, instead of isolation. An example of the potential of these brain-inspired network models is given in Appendix A, and discussed in the papers of Capecci et al. (2015), Chen, Hu, Kasabov, Hou and Cheng (2013), D. Taylor et al. (2014).

It was originally intended that the NeuCube use the pLIF neural model (*cf.* Section 3.3.6). At present, all explorations of the NeuCube have been performed with the standard LIF neural model. In the case of the generic model, it is not important which neuronal model is chosen. In a specific case, this selection may matter, most

particularly when we implement the NeuCube on neuromorphic hardware systems which have limited neuronal models available.

4.3 Output Classifiers

After training the NeuCube reservoir, we must then classify or otherwise create some meaningful output from the reservoir behaviour. The spiking trajectories of the reservoir neurons are meaningful, and recorded over time represent the transformed input data. We train a classification, regression, or other output system on these spike train recordings. Kasabov (2012b) and Kasabov (2014) suggest the deSNN and SPAN learning algorithms previously discussed for use in this context. Any suitable technique can be used here, including the learning systems discussed in Section 3.6.2, and a number of traditional machine learning systems such as the SVM.

Specific methods of output for the NeuCube will not be discussed here. These will follow the principles established in Sections 3.6.2 and 3.6.3. Considerations for their use in the NeuCube framework and in the context of our desired application are discussed in Section 5.3.

The definition of the NeuCube framework in Kasabov (2012b) established the possibility for recurrent connections from the classifier to the reservoir. This is feasible only in on-line learning - as otherwise, the reservoir naturally has no ability to adapt to these recurrent connections as it is no longer active – and has yet to be implemented.

4.4 Neuro-Genetic Optimisation Network

The NeuCube framework also incorporates an Computational Neuro-Genetic Model (CNGM). It was originally planned that the CNGM introduced in Section 3.6.3.3 would either interface with or replace these neuronal models. The CNGM feature has yet to be implemented; a future development will incorporate this feature. Considerations for the design of the CNGM are discussed in Section 5.4.

4.5 Visualisation Technologies

While not explicitly a part of the NeuCube framework, the facility for visualisation of the complex networks created is vitally important. Whether it is used for behavioural analysis and optimisation, or knowledge extraction from the fully evolved model, a key component of the NeuCube is the discoverable and transparent memory and

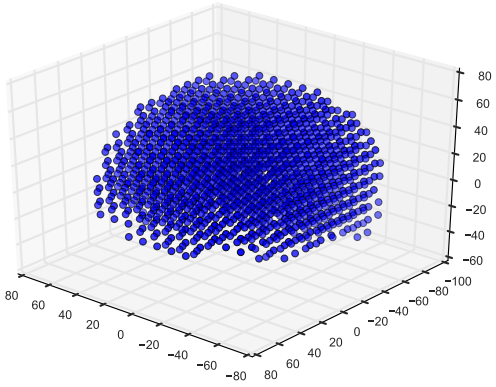
neuronal dynamics. This transparency is a significant advantage over many other state-of-the-art machine learning approaches to SSTD, as these are typically treated as a 'black box', or are otherwise structurally meaningless to an external observer. These visualisation tools can be described in a little depth here, as they are a contribuition of this thesis that is not directly related to the primary research questions identified in the introductory chapter. Visualisation of these networks as a tool should be performed in concert with the signal processing and statistical techniques commonly used in the interpretation and analysis of SNN models, and are not intended to replace these. They are however, intended to augment them, as in the case of the NeuCube we have the ability to meaningfully interpret changes to the network structure. This feature of the model is enabled by our use of a network structure that references the data collection context, and the creation of connectomes which facilitate the communication of coupled spatio-temporal signals in our input data.

The value in this feature of the NeuCube is that it allows for a richer interpretation of the visual representation of the trained models. Here, we can more easily identify anomalous behaviours or general trends visually, which may not be apparent in statistical or signal-processing based interpretations of the model. Additionally, it provides an intuitive entry to the analysis of such models for non-expert users, such as neuroscientists or geoinformaticians. In Section A.2, the trained connectome was interpreted by a neurologist who identified that the model had generally replicated the weighting of the brain areas used in the generation of the input signal. This could in theory then be used to interpret the differences between model responses to diseased or control patients, or identify anomalous processing pathways not normally apparent in such models.

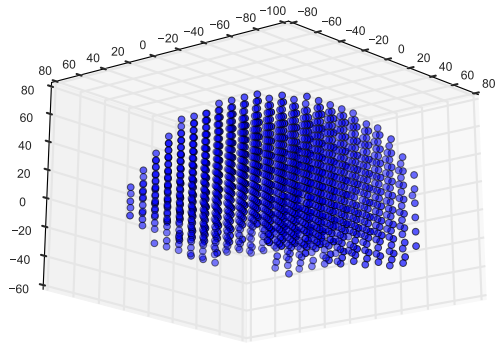
Two levels of visualisation system have been developed for the NeuCube: the inbuilt simulation visualiser, and the external immersive visualiser.

4.5.1 Standard Visualisation of the NeuCube

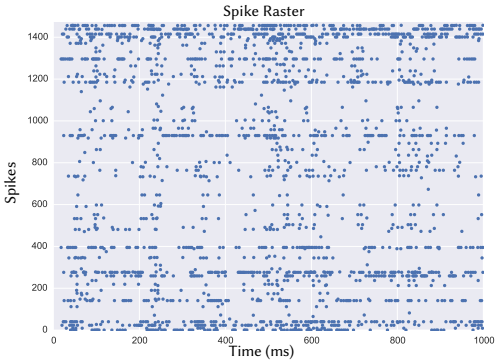
Incorporated in both the NeuCube implementations discussed later in this thesis and the prototype M1 version, is a basic set of visualisation tools. We discuss these only briefly; they are not particularly novel from a computer graphics or visualisation perspective, but are still a necessary component of the knowledge extraction process, and novel in the context of SNN. These visualisation tools are supported by the visualiser introduced in the next section, which is more dynamic and meaningful in the context of temporal data.



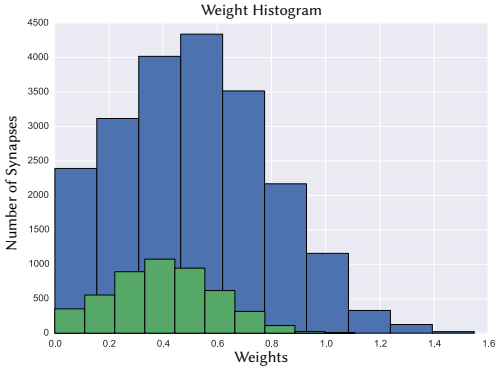
(A) 3D reservoir network structure visualised from above.



(B) 3D reservoir network structure visualised from below.



(C) Raster plot of spike timings per neuron in the reservoir



(D) Histogram of synaptic weights after training

Figure 4.2: Examples of the basic visualisation system developed for the implementation of the NeuCube in PyNN introduced here. Basic plots supported include visualisation of the 3D network structure (Figures 4.2a and 4.2b), spike timing raster (Figure 4.2c), and synaptic weight histograms (Figure 4.2d).

Visualisation of the reservoir structure and connections is provided through an orthogonal 2.5D plot, which can be moved and navigated through with the mouse in a relatively intuitive manner. In the case of the implementation of the NeuCube introduced in this thesis, the visualisation engine we have developed uses Python's `matplotlib` and `seaborn` libraries. Additional plots, including spike rasters and weight histograms, have been developed, and provide the core of the knowledge extraction process. Plotting in the PyNN version can be automatically shown after a simulation, automatically saved, or a combination of the two. This automated plot saving is intended for use in unattended simulations, such as when the model is run remotely on a cluster. Examples of these plots are shown in Figure 4.2.

These tools have been used directly in the validation and analysis of the NeuCube models developed in Appendix A and B. In particular, in Appendix A and Figure A.1, it is shown that a trained NeuCube model, when properly intialised and visualised,

can demonstrate a connectome which represents a biological phenomenon with some accuracy.

4.5.2 Immersive Visualisation of the NeuCube

With a system like the NeuCube, where the 3D structure and behaviour over time is significant and meaningful, the amount and complexity of the acquired data necessitates new forms of visualisation to allow users to extract meaning and structure from it (Bruckner et al., 2009; Lin et al., 2011). In these cited examples, effective visualisation enables navigation in the dense 3D space, and selective rendering of regions, pathways, and structures. However, the render is confined to the 2D screen of the host computer. Navigation within the 3D space is mediated by mouse and keyboard controls, providing only an indirect perception and a reduced representation of a rich 3D space. Similarly, the basic visualisation tools discussed in Section 4.5.1 limit the amount of useful structural and temporal dynamics that can be extracted from a NeuCube model. To address this issue, in Marks, Estevez and Connor (2014) and Marks, Estevez and Scott (2015), we introduced an immersive and intuitive truly 3D visualisation system. In this section we review this contribution. Interested readers are directed to the cited papers, particularly Marks et al. (2014), for further details.

Through the use of stereoscopic Head Mounted Displays (HMDs), a sub-millimetre precise motion tracking system, and a bespoke rendering system, Marks et al. (2014) and Marks et al. (2015) present a framework incorporating both hardware and software[2], for natural human gesture interaction with an accurate 3D visualisation incorporating depth cues. Users can control the system using natural gestures. Navigation within the space is performed by walking, the render view by the user's head position, and selection of visual cues by pointing with a glove-based marker system.[3] A haptic interface device for the system was introduced in Foottit, Brown, Marks and Connor (2014), which provides the facility for the user to receive natural, tactile feedback from the system.

This natural interaction is in contrast to the existing visualisation systems, such as those in Bruckner et al. (2009) and Lin et al. (2011), which require the use of a keyboard and mouse controller. Our system can also incorporate these control schemes, as

[2]Private repository forked to https://github.com/nmscott/NeuVis. For access please contact the author or Dr. Stefan Marks of AUT's CoLab.

[3]Video of the visualisation being used in the AUT MoLab motion capture space can be seen at https://youtu.be/NJVxYdvM-W8, https://youtu.be/sJND8jp9QgU, and https://youtu.be/MsuPp3D4iaw.

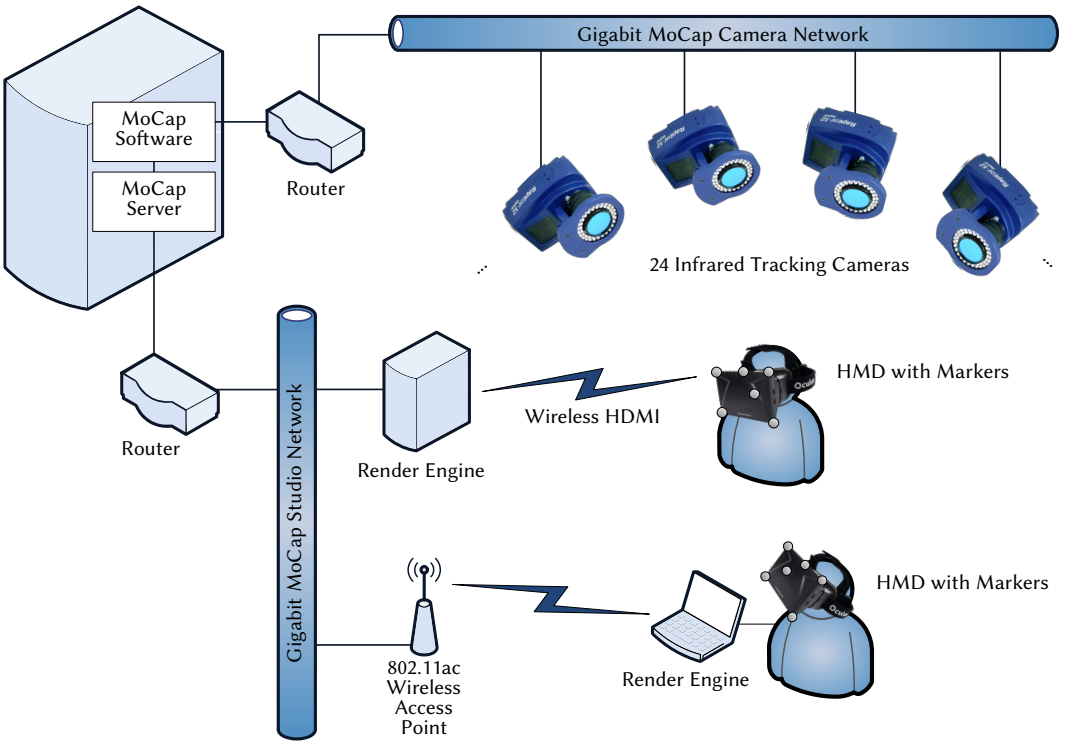


Figure 4.3: Immersive Visualisation system architecture overview. The 24 tracking cameras send positional data back to the render engine, which generates the stereoscopic 3D visualisation data and sends it wirelessly to the HMD. Figure adapted from Marks, Estevez and Connor (2014).

well as standard gaming control pads, if the space is limited and the motion capture systems are not available. More similar to our system is the work of von Kapri, Rick, Potjans, Diesmann and Kuhlen (2011) which uses a Computer Assisted Virtual Environment (CAVE) to visualise the spatial structure and activity of a spiking neural network. Again, natural navigation is not possible in this environment, and concurrent users are not possible. Our system can support an arbitrary number of users, provided that sufficent hardware (HMD, markers, render nodes) is available.

The Immersive VR Space, shown in Figure 4.3 comprises the following major components:

1. A wide area motion capture suite,
2. Head Mounted Display (HMD), and
3. Render engines.

The motion capture system uses infrared optical markers, captured through 24 cameras mounted in a capture volume of 6 m × 6 m × 3 m. These cameras provide

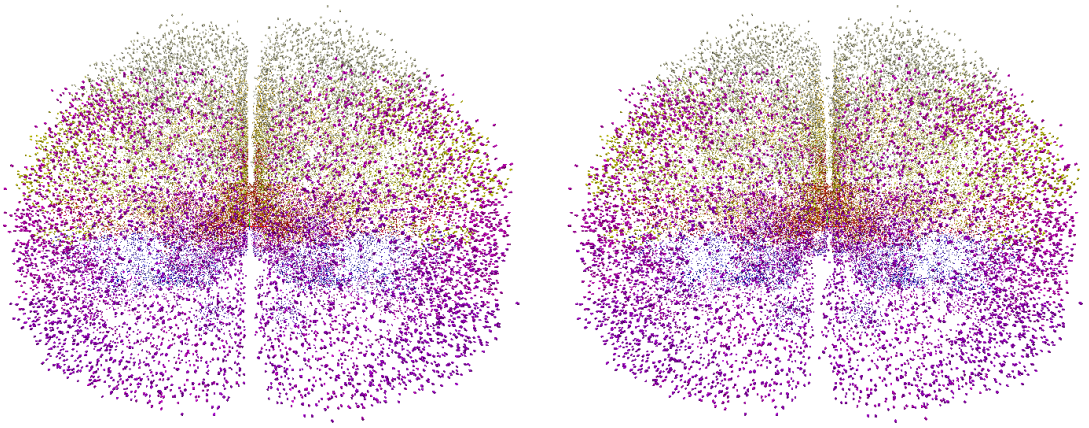


Figure 4.4: Stereographic view of the Talairach atlas as being displayed within the HMD of the user. The leftmost figure is displayed in the left lens, and the rightmost in the right lens, giving an illusion of depth. Note that the background is white for print purposes. The actual visualisation uses a dark textured background pattern. Figure from Marks, Scott and Estevez (n.d.).

positional accuracy of up to 0.1 mm at 180 fps. Infrared is used to minimise interference from natural lighting conditions, and the high number of cameras is required to ensure that accurate positional information can be inferred regardless of the subject's pose. Marker positions are captured and localised into the space by a motion capture server, which data is then broadcast to the render engines. These render engines then create scenes based on this position and orientation information, combined with the position and orientation of other significant markers such as pointer devices (in our case, a glove with tracking markers). A more comprehensive discussion of this architecture is given in Marks et al. (2014).

Presently, the system is capable of rendering up to 1.5 million neurons and their connections with a steady framerate of 60 fps. See Figure 4.4 for an example of what is rendered here. The view frame is divided between the two brain shapes, which are offset sufficiently that when rendered separately in the two lenses of a typical HMD they show a true 3-dimensional view of the network. Individual neurons can be selected using the hand-based cursor shown in Figure 4.6, which shows visual and textual information about the selected neuron; *e.g.* type, potential, incoming and outgoing connections. For an example of what is visualised, see Figure 4.5. This allows us to analyse the behaviour of single neurons or populations as a whole, within the same space and using the same tools.

The capacity to render a large number of neurons and connections efficiently allows us to visualise and interpret large scale networks which would be otherwise

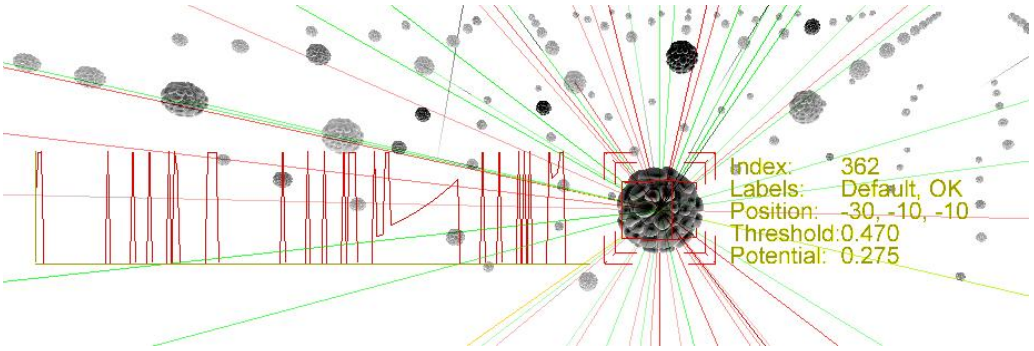


Figure 4.5: A hand-based cursor node can be used in a natural way to view additional information about a specific neuron and its activity in the reservoir: for example, internal membrane voltage, spiking histories, *etc.* Figure from Marks, Estevez and Scott (2015).

impossible to view in standard visualisation tools. This allows us a greater potential understanding of the general patterns and behaviours of such networks than would otherwise be available. As yet, the networks developed in the NeuCube have been small (< 5,000 neurons), so standard visualisation tools such as those introduced in Section 4.5.1 are still sufficiently effective.

While as yet a systematic study has not been conducted, so far, around 50 visitors of the Immersive VR space have experienced this visualisation. It was reported in Marks et al. (2015) that we have observed in general, people quickly start to move around and look at structures and point out individual neurons using the 3D cursor. We have received positive feedback about the appearance of the visualisation and that interaction metaphors are very intuitive. This system has also been used to discover some small bugs or inconsistencies in alternative mappings of the NeuCube reservoir, such as that in Capecci et al. (2015) where input neurons were being inadvertently duplicated over the lifecycle of a single simulation.

The only larger issue is related to distortion of the field of view and different ratios between actual motion and "visible motion". When users move the hand cursor, experience and muscle memory provides a different "perceived position" of the cursor than is visible in the HMD. This issue, however, can be adjusted by a more careful selection of render parameters and distortion correction for the lenses of the particular HMD chosen.

It should be noted here that this visualisation system is not limited to use in the large motion capture space. It is equally useable when in a standard lab space. The motion capture system can be replaced with the simplified motion tracking systems

Figure 4.6: A user navigating through the virtual representation of the NeuCube network, using an intuitive, hand position based 3D cursor. Markers on the HMD and glove are translated to position and orientation, and used to localise the user within the render space. Figure from Marks, Estevez and Connor (2014).

incorporated in most modern HMDs. The intuitive gesture-based control system can be replaced with either a gaming controller or a keyboard and mouse combination. Additionally, this visualisation is not limited to true 3D stereoscopic rendering; it can as easily be rendered in pseudo-3D on a standard desktop monitor. This allows us to use the system in different environments, and does not tie our interpretation of a complex model to the availability of a state-of-the-art motion capture suite.

It is expected that going forward, as the networks implemented in the NeuCube grow in scale and complexity, that this system will be applied more directly. It should eventually supplant the basic orthogonal visualisation tools incorporated in the various NeuCube packages, although with the assumption that basic plotting functions such as spiking rasters will always be available.

4.6 The NeuCube Framework in Practice: Design, Implementation, and Applications

In this chapter, we have introduced the basic concepts of the NeuCube framework, as it was initially defined in Kasabov (2012b) and Kasabov (2014). The NeuCube is an

adaptive and novel SNN framework, designed for the analysis and interpretation of SSTD. This introduction was abstract; we introduce details of these concepts in later chapters.

The question at this point, then, concerns how this framework can be applied in practice. How do we design a NeuCube? How should it be implemented in software or hardware, and is there some empirical evidence that it is advantageous over traditional techniques?

In the successive chapters, we resolve these questions. Firstly, we discuss a general design methodology for the NeuCube in Chapter 5; *i.e.*, which components and parameters should be used given some context, and how these may affect each other. Chapter 6 introduces a general software framework and architecture for the implementation of the NeuCube, and a concrete implementation of the same using the PyNN library. Subsequently, in Chapter 7 this software framework will be adapted for implementation on different neuromorphic hardware devices, with a particular emphasis on the SpiNNaker system. Empirical evidence in the form of two proof-of-concept case studies are discussed in Appendices A and B.

Contributions of this Chapter

1. A review of the already established NeuCube architecture

Peer Reviewed Literature With Direct Contributions from this Chapter

1. Kasabov, N., **Scott, N. M.**, Tu, E., Marks, S., Sengupta, N., Capecci, E., Othman, M., Doborjeh, M., Murli, N., Hartono, R., Espinosa-Ramos, J.I., Zhou, L., Alvi, F., Wang, G., Taylor, D., Feigin, V., Gulyaev, S., Mahmoud, M., Hou, Z.-G. and Yang, J. (2016). Evolving Spatio-Temporal Data Machines Based on the NeuCube Neuromorphic Framework: Design Methodology and Selected Applications. *Neural Networks*. Special Issue on Learning in Big Data. Elsevier. doi:10.1016/j.neunet.2015.09.011
2. Sengupta, N., **Scott, N. M.**, and Kasabov, N. (2015). Framework For Knowledge Driven Data Encoding For Brain Data Modelling Using Spiking Neural Network Architecture. In *5th International Conference on Fuzzy and Neural Computing*. 17–19 December 2015. Hyderabad, India. Springer. doi:10.1007/978-3-319-27212-2_9
3. Marks, S., Estevez, J. E., and **Scott, N. M.** (2015). Immersive Visualisation of 3-Dimensional Neural Network Structures. In *13th International Conference on Neuro-Computing and Evolving Intelligence*. February 19–20, Auckland, New Zealand.
4. **Scott, N. M.**, Kasabov, N., and Indiveri, G. (2013). NeuCube Neuromorphic

Framework for Spatio-Temporal Brain Data and Its Python Implementation. In *Proceedings of the 20th International Conference on Neural Information Processing*, 3–7 November 2013, Daegu, Korea. Springer. doi:10.1007/978-3-642-42051-1_11

Chapter 5

Design Methodology of SNN based on the NeuCube Framework

> Every man can, if he so desires,
> become the sculptor of his own brain.
>
> — Santiago Ramòn y Cajal
> (*Recuerdos de Mi Vida*, 1937)

A meaningful contribution of this thesis is the development of a methodology for the *design* of arbitrary NeuCube implementations. In this chapter, the key considerations for design of a NeuCube application in will be raised in context.

The design considerations introduced here (bar the discussions regarding the 3D structure of the network) are applicable and generalisable to *all* SNN systems. They are introduced here in the context of the NeuCube framework, but are equally relevant when considering alternative SNN architectures. In particular, the considerations for encoding systems and output classifiers are of value to any generic SNN model. In the absence of a robust information theory of SNN, the design process for NeuCube architectures must be heavily based on empirical observations and heuristics. Here, we attempt to develop a meaningful and effective methodology based on these heuristics.

We inform our design decisions by two primary factors:

1. What *a-priori* information do we have about the data?

And, subseqently:

2. What is our intended application?

Some specific considerations need to be made in the design of such systems depending on the computational platform chosen. See Section 5.5 for a brief discussion of this topic, or Chapters 6 and 7 for comprehensive discussions on considerations for software- or hardware-based simulation, respectively. When interpreting the following discussions, the reader should bear in mind the context of the computational device used; *e.g.* when utilising a neuromorphic VLSI ASIC (*cf.* Section 7.3) which is physically limited to the AdEx neuronal model, we should not make a decision in the reservoir design to use the pLIF or Izhikevich type neuronal models.

5.1 Encoding Systems Design

It is not possible to state with certainty what the most efficient spike encoding method is, as this is inextricably linked with the context of the simulation and the type of data used. Furthermore, it is heavily dependent on the intent behind the term "efficient"; here, do we mean the most compact representation? The fastest to process the input data? The most effective in terms of the output quality? These decisions cannot be made in isolation, as they affect the system as a whole. Additionally, there is no body of literature to indicate this with any theoretical rigour. For these reasons, it is not possible to define the most efficient spike encoding. This is, at present, a heuristic process.

In order to select an effective encoding method, we must consider the data.

1. What are the characteristics of the data are we encoding?

In this we identify the context of the data. Is it high-frequency, with a low number of channels? Is it streaming or fixed? Is there some known representation which we can utilise? The rate and nature of the incoming data will dictate which encoding method is optimal – given that the next question is resolved.

2. Rate or temporal coding?

Do we intend for the spiking times of the encoded data to be temporally significant, or are we content with rate coding? These different forms of encoding are appropriate in different contexts. While the NeuCube is typically considered to use the temporal coding paradigm, it is also capable of rate-based input. Unless we have a specific and deliberate reason to use a rate-coding scheme, we would default to a temporal coding scheme. One notable exception to this rule is when we wish to use the BSA

algorithm for the encoding of EEG data. Depending on the context this may be more appropriate than a temporal-coding method.

In a general sense, the Threshold-Based Temporal Difference (TD) is useful for most types of data, given that appropriate parameters are selected. The threshold rate and step size of this algorithm is particularly vital in effectively encoding data. The optimal value for these variables should be possible to derive analytically: for example, the threshold by setting this value to some proportion of the average statistical variance of the input data, and the step size by some proportion of the average internal frequency of the data using the Nyquist-Shannon theorem. This is an area which we intend to address in the future, as the behaviour of encoding systems is not well researched.

Other forms of encoding are more appropriate for different data contexts. As mentioned, both the TD and BSA encoding schemes have been shown to be effective for certain EEG data. The Knowledge Driven Data Encoding scheme has been shown to be effective on fMRI data, particularly when the issue of signal reconstruction is concerned. Population Encoding may be more approriate when we are dealing with static (real-valued) data.

5.2 Reservoir Design

The design of a NeuCube reservoir is very much a heuristic process. However, it is not done blind; a number of factors based on the nature of the data influence the decisions made during this process. Primarily, we must first consider:

1. What *a-priori* information do we have about the nature and structure of the data?

This informs the design of the NeuCube reservoir's 3D structure, its connectome, and the locations for data to be input.

In general, our primary heuristic is that the reservoir should represent the inherent spatial or spectral charateristics of the data as well as possible; in doing so, it will retain the vital temporal links within the data, and the implicit information contained within them.

5.2.1 Reservoir Topology Design

2. How can we best represent the data source with the reservoir topology?

This question, in essence, asks what network structure we should use, based on our understanding of the properties of the data. Of most interest at this point is any inherent structure in the data collection context. In this, it is meant – is there some physical representation which best captures the spatio-temporal dynamics of that data? In the case of neuroinformatics data collection, this would likely be a brain shape, as this best represents the physical properties causing the temporal dynamics of the data.

The emphasis here is on maintaining the representation of the physical reality of the data in the model. In this way, we can retain the vital spatio-temporal links inherent in the data, which in many cases are the most significant property of that data. As a corollary of this, the retention of a 3-dimensional structure over an n-dimensional model ensures that we can intuitively analyse the network structure and evolution through visualisation tools. An n-dimensional network structure would not explicitly represent the input data space in most cases. This type of NeuCube has not been explored in the literature as yet, due to the initial emphasis on the explicit structural representation of real-world data in the NeuCube reservoir. In the case that there is no inherent structure to the data, we can structure the model in an arbitrary way, with no meaningful shape. It is often possible to provide a known architecture, based on the context of the data.

For example, in the case of a neuroinformatics task, there are a number of established stereotactic altases which map known brain locations or features to positions in 3D space. One such example is the Talairach atlas introduced in Talairach and Tournoux (1988), which is commonly used in MRI and EEG data collection. As we know this atlas reasonably represents the volume and structure of the human brain in space, it is therefore logical for us to use this information to structure the NeuCube reservoir. The greater volume of the Talairach space can be transformed to a point cloud, where the points represent reservoir neurons in that space. This point cloud can have an arbitrary resolution; the highest resolution Talairach atlas at present contains about 1.4 million points each representing around 1 mm^3 of brain volume. Our most common NeuCube structure uses 1471 neurons representing about 1 cm^3 of brain volume each. The resolution here – *i.e.* the number of neurons in the model – is adaptable depending on the data source and application context. In this example we have developed a model where each neuron represents 1 cm^3, as this is the rough resolution of the EEG device selected. Most EEG devices have the capacity to collect signals from around 1 cm^2 of scalp area, with the penetration depth around the same depending on the device and task type (D. Taylor et al., 2014). Equally, if we were to

use a high-resolution fMRI input signal, our spatial resolution could be increased up to 1 mm^2 per neuron or beyond. This selection is highly contextual, and has not yet been formalised. Here, we must make some heuristic decisions based on the data and the overall performance of the system.

Similarly, in the case of seismic data, we could use some map representation of the seismograph sites, with distances in the network set proportionally to the distances in the real world.

If there is no meaningful spatial component to the data collection context – for example in the case of radioastronomy or financial time-series data – there may be some *spectral* structure which we can conceptually represent in the reservoir. In this case the reservoir structure is less significant. Here, we should ask what the underlying structure of the data is, and how this might best be transformed into a spatial representation. Is there some characteristic to the data collection context which may be of interest?

In the case that there is no known underlying structure to the data, we can structure the reservoir arbitrarily. By this, we mean that instead of a meaningful 3D representation of the data collection context, we can use an arbitrary shape (*e.g.* a $10 \times 10 \times 10$ neuron cube) as we would normally do in the reservoir computing paradigm. It is worth noting that in this case, the actual representation of the data in the model may be more meaningful than that of an LSM or typical reservoir computing architecture due to the explicit encoding of the data structure into the NeuCube's input topology and connectome (Kasabov, Hu et al., 2013).

The neural model chosen is dependent on the computational platform and the particular application area. Generally, we should avoid microscopic neuron models like the Hodgkin-Huxely in favour of more concise engineering formulations like the LIF or Izhikevich models, as the extra biological accuracy is not necessary and computational efficiency is more beneficial.. Section 4.2 mentioned that the original intent of the NeuCube framework was to use the pLIF neuronal model. It is expected that the pLIF will improve the NeuCube reservoir's non-linear pattern separability, in line with the results reported in Schliebs et al. (2010) and Nuntalid et al. (2011). At present, we have found acceptable results with the standard LIF model neuron. It is worth trialling the pLIF if the standard LIF does not provide sufficient performance on your particular dataset; however, in this case, we must be aware of the caveats of this model. The use of the pLIF is not typically computationally feasible when utilising a neuromorphic hardware simulation platform. In the case of the SpiNNaker, it was

previously too computationally costly to implement Random Number Generation (RNG) in software on small devices, as the chip architecture used does not incorporate hardware for this (*cf.* Section 7.4.2.4). Subsequent to the initial development of this thesis, a computationally efficient RNG has been implemented in software, which has therefore enabled the use of pLIF model neurons. In the case of systems such as the VLSI ASICs introduced in Section 7.3, it is not possible to change the neuronal model as this is a physical property of the circuit. This issue is discussed more in depth in Section 5.5.

Typically we would have around 20% inhibitory neurons in the reservoir, as is standard practice in reservoir computing. The vast majority of computational platforms support some form of inhibitory neuron or synapse. A lack of inhibition in the network can cause it to 'avalanche' and saturate, spiking as rapidly as the simulation system can process. This is an undesirable behaviour – in biological networks, this is the cause of certain types of seizure – and should be avoided. If inhibition is not possible in a simulation, as a mitigation strategy the use of an adaptive neuron model such as the AdEx should be considered. In this case, the neuron's spiking threshold automatically increases as its spike rate increases, self-regulating to an extent.

The resolution of this reservoir (*i.e.*, the number of neurons per unit of space and number of synapses per neuron) is generally set heuristically, as there is not yet any significant information theory for SNN. Recall that larger networks, and more particularly networks with a high number of synapses, are generally considered to have better ability at handling complex patterns, and greater memory capacity. However, with the meaningful structures introduced in this thesis, smaller networks are also effective in complex applications. Additionally, with these meaningful structures, we can in some cases infer the network resolution from the data itself. Consider fMRI; this data source represents a temporal series of 3D brain volumes. This volume is comprised of voxels – 3D pixels – which have a certain resolution. We can therefore draw some idea of the network structure from this voxel set, representing each of these fMRI voxels with one or a small number of neurons in the simulation. This is consistent with the biological reality generating the data, as one voxel represents the collective activation of a number of neurons in that particular location. In this way, we ensure that the reservoir represents the underlying relationships within the data as best we can, which retains the implicit information in these relationships.

In reality, network resolution is a tradeoff between computational cost and the complexity of the data; typically we should err on the side of larger networks in order

to exploit their advantages, provided that the computational cost does not negatively impact on the system's real-world effectiveness. Smaller networks are of course, better optimised, and so in the case of low-power or high-speed applications, we should attempt to find the smallest functional network possible. In certain cases it may be acceptable to sacrifice some effectiveness for a smaller network, particularly when implementing systems in neuromorphic hardware.

5.2.2 Input Topology Design

3. How can we best represent the structure of the input data in the reservoir topology?

The issue of input location design is similarly difficult. In essence, we must ask if there is some known or meaningful mapping of the input locations, within the space created by the reservoir?

For example, consider the case of EEG. This particular architecture is discussed in more depth in Section A.1. The location of the collection (EEG) device is based on a well-known standard template which can also be associated to a known cortical location in the Talairach or Montreal Neurological Institute (MNI) spaces. In the case that we have used one of these spaces to structure our reservoir as discussed in Sections 5.2.1 or A.1, these two spaces are easily reconcilable. A known physical data capture point in the real world can be conceptually mapped to its equivalent location in the reservoir in this way (D. Taylor et al., 2014).

The location of these inputs, therefore, is contextual. From our understanding of the type of data used and its collection context, we should be able to develop a mapping which retains the positioning of the input in the reservoir proportional to their relationship in the real world. In this way, our model can explicitly encode the spatial context of the signal, informing our interpretation of the output data and the evolution of the model as it learns (Kasabov, Hu et al., 2013; Kasabov et al., 2015). It is possible here that there is a more advantageous n-dimensional structure for the reservoir. High-dimensional reservoir structures have not been examined in the existing NeuCube literature, as the intent of the system is to model the reservoir in as close to a natural context as possible to facilitate knowledge capture and interpretability of the trained models. This is a valid area of exploration in future, and should be considered where a natural 3-dimensional mapping is not possible or feasible.

We must also consider the structure and resolution of the reservoir when generating our input locations. In the case of fMRI, the number of voxels in a high-resolution image can be upwards of 100,000. If we are attempting to map this into a smaller network (*e.g.* the typical 1,471 neuron Talairach-based reservoir), obviously some data merging must be performed prior to a mapping being created. In a case such as this, it is possible to combine a number of input channels whether through averaging or some statistical measures. This combined input channel can then be represented by one reservoir input location. Dimensionality reduction is to be avoided where possible. If performed, it should be done with awareness of the data context. The so-called 'curse of dimensionality' – various phenomena arising from the analysis and processing of data in high-dimensional spaces – is largely avoided in the NeuCube, as despite the fact that the reservoir processes data in a high-dimensional space, our input and outputs are generally of limited dimensionality. Additionally, if issues of dimensionality are apparent in the behaviour of the NeuCube, its reservoir size or number of dimensions can be increased to mitigate this.

In the case of purely spectro-temporal data, the issue of mapping input to reservoir space is more complex. For example, in the case of the radioastronomy task discussed in Appendix B, there is some characteristic shape of the spectro-temporal data. Here, the frequency bands are dispersed in an inverse-quadratic shape due to delays in the lower frequency bands caused by interstellar media. We can represent this structural characteristic of the spectral data in a physical form, by creating input locations in the reservoir to physically match this dispersion pattern in 3D. This ensures that the structural links – though they may not necessarily be *spatial* links – in the data are retained, preserving the timing of the original signals and with it, the implicit information of those timing patterns.

5.2.3 Connectome Design

We must now make some design decisions regarding the synaptic matrix of the reservoir. This synaptic matrix is responsible for the spike communication within the network, and its evolution provides the basis for the unsupervised learning.

4. What connectome should we use?

In almost all cases this will be the 'small world' structure discussed below. However, certain cases we may have some *a-priori* knowledge of the network structure we should use. For example, in the case of a neuroinformatics task, we may have DTI data.

Neurons in cortex are commonly connected with what is known in graph theory as 'small world' connections (Stam, 2004). In most cases, this is modelled as a simple lambda-distance dependent probability, where closer neurons are more likely to be connected and more distant neurons are less likely to be connected. Small world interconnectivity of neurons can be calculated with

$$P(c_{ij}) = C \times e\,\frac{-d_{ij}}{\lambda^2} \tag{5.1}$$

as detailed in Maass, Natschläger and Markram (2002), Nuntalid et al. (2011), where $P(c_{ij})$ is the probability that a connection will be made between neurons i and j, d is the distance between those neurons, C is a constant, and λ is a factor controlling the probability of a connection. Due to its biological plausibility and effectiveness in prior experiments, this connection structure is appropriate for use in this thesis. This basic connection generation method is discussed further in Section 6.4.3.1, albeit with some small modifications for ease of implementation.

If we have some additional information about the structure of the network, it may be possible to extrapolate a connectome from that. As briefly mentioned, in the case of Diffusion Tensor Imaging (DTI), we can directly generate a network connectome from this. DTI is a measure of fluid diffusion in the brain, which indicates the direction and mass of white matter (*i.e.*, the connections) in that volume. Essentially, DTI provides us a list of 3D locations within a brain volume and the primary direction of the connections in that area. As this is typically registered to the Talairach or MNI spaces, it is therefore possible for us to use the actual connection structure within a person's brain to generate a NeuCube reservoir's connectome. In this way, not only can we keep the physical locations of inputs and neurons within the same space, we can ensure that the connections between these areas represent – to the best of our ability – the physical reality of the original connections. Note that the connectivity in the brain is typically considered to be small-world (Stam, 2004) so even without this informed connectome generation, the default probabilistic process will still yield a meaningful network structure.

The actual synapse dynamics will primarily be dependent on the computation platform used. This consideration will be discussed in Section 5.5.

5.3 Output Classifier Design

This section is entitled 'Classifier' design as this is the most common form of output for the NeuCube, although the considerations here will largely be common across all

output devices, regression or prediction included. Naturally, the first question in the design of the output device, then, is:

1. Classification, prediction, or regression?

The intent of this question should be clear. From this, we branch into three different areas of output device: classifiers intended to group and cluster data, and respond to these clusters in some defined way; predictors, intended to classify data prior to all of the information being available; or regressors, intended to extrapolate from the data in some way. This question is directly related to the second:

2. What is the context of the application?

Do we intend to use it for adaptive classification of fast moving vision data? Smooth motion control of a robotics device? Simple classification of financial data? Our response to this question defines the type of classifier or predictor algorithm chosen. It has less applicability to regressors.

The actual learning mechanism should be chosen pragmatically, based on the intended application of the NeuCube model we are developing. SNN classification methods have been initially suggested here as they will retain the temporality of the reservoir response, and process it in the same manner. This is advantageous when operating on highly temporally-dependent input data, but may not be necessary for simple offline classification tasks. In this latter case, more traditional machine learning techniques such as the SVM applied to a time window of the reservoir response may be appropriate. There is not yet a formalisation of the optimal selection of output algorithm, as it is highly contextual based on the data and the desired application (Kasabov et al., 2015). The SNN learning methods discussed in Section 3.6.2 are generally applicable to most NeuCube models, with specific selections being based on the properties of the method. For example, in the case that we want adaptive, predictive classification of fast-changing data, we could implement the deSNN algorithm, utilising a low C value. For smooth control of a robotics device (*i.e.*, precisely timed output spikes), we could implement the SPAN learning algorithm. Our selection of classification algorithm should be based on the desired output characteristics.

5.4 Gene Regulatory Network Design

The most reasonable question to ask at this point is whether we even need to use the CNGM in our model. At present, the CNGM is not implemented and without it, the NeuCube has shown some remarkable results – including those presented

in Capecci et al. (2015), Chen et al. (2013), J. Hu et al. (2014), Kasabov and Capecci (2015), Kasabov, Hu et al. (2013), Kasabov et al. (2015), Scott et al. (2013) and D. Taylor et al. (2014). Therefore, we must first ask:

1. Do we need to incorporate the CNGM?

The application of a CNGM is of more importance in the case of brain data, as discussed below. In the case of general data (*i.e.* anything other than brain data in this context) the CNGM would only be meaningful as an automated optimisation system; used in this way, a more efficient system such as the quantum-inspired PSO of Schliebs and Kasabov (2013) may be more effective at lower computational cost.

2. Can we use some *a-priori* information about the gene expressions?

The CNGM simplifications introduced in Section 3.6.3.3 are valid for earlier forms of GRN and CNGM. For our purposes however, we are able to introduce some sophistication to these assumptions. In particular, with the assumption that we are studying brain data with a brain-shaped reservoir, we are no longer bound by each GRN having the same initial point. Using an appropriate gene ontology, we are able to map specific gene expressions and therefore, protein levels, to a spatial location in the CNGM. This is contended to be a significant advantage over the current state of the art, as it should lead to a less homogenous model and higher non-linear pattern separation ability.

Real gene data obtained by microarray analysis should be used in CNGM to address a number of issues, including introduction of the capacity for exploration of addition and deletion of genes within the model, or its use for modelling neurodegenerative diseases (Benuskova & Kasabov, 2008). A useful source of this gene data, appropriate for this thesis, is the comprehensive Allen Brain Atlas[1] (Hawrylycz et al., 2012). This atlas contains the averaged gene expressions of a large number of donor brains, with genes associated with the physical location in the brain that they are expressed.

Useful knowledge can be extracted from networks modelled in this way. For example, in an SNN-based computational model of epilepsy, the desired epileptic behaviour is achieved by a biologically inspired moderation of the network parameters controlling neuronal inhibition ($GABA_A$ or $GABA_B$ expressions for example) (Kudela, Franaszczuk & Bergey, 2003; Wendling, Bartolomei, Bellanger & Chauvel, 2002). The same could theoretically be done for most genetic brain impairments, including degenerative diseases such as Alzheimer's and Parkinson's (Kasabov et al., 2011).

[1] Available from http://human.brain-map.org

CNGM design becomes more difficult when we are no longer using the brain-shaped reservoir. Without the biophysical cues and *a-priori* information from sources like the Allen Brain Atlas, where do we express which genes? Here, we must make some heuristic decisions. If we are integrating a multimodal source, we could choose to more strongly express inhibitory genes in an input area which is expected to see high-frequency data, and excitatory genes in an area expected to see low-frequency in an attempt to balance the network. Over a long enough time-frame, the CNGM should eventually reach some form of homeostasis regardless of its initial starting point.

5.5 Influence of Computational Platform on Design

Our choice of computational platform – that is, the simulation platform used to model the behaviour of our model – will directly affect a number of factors in the design of NeuCube models. The use of software-based simulation provides a different set of constraints to the use of a neuromorphic hardware simulation.

Here, we very briefly discuss some of the major considerations we must make when developing models on different computation systems, and in particular, some specific considerations we must make when developing for mixed-platform simulation. By this, we mean those models we intend to develop to run on heterogeneous simulation systems, such as a model trained in software which is then run on a SpiNNaker board.

The relative merits of each approach will not be addressed here – this discussion will focus specifically on those constraints that directly affect the design of SNN models, particularly those that affect the NeuCube framework. See Chapters 6 and 7 for a discussion on the nature of software- or hardware-based simulation, respectively.

5.5.1 Software Based Simulations

In broad terms, software based simulation systems offer the user greater flexibility in terms of model design. Scaling of network size, choice of neural or synaptic model, and reconfigurability are all significantly eased when developing for a software-based simulation. Network scaling has been stated as an advantage in software; in this, the ease with which we can define a larger or smaller model is meant. In reality, neuromorphic hardware systems are advantageous when scaling due to their power consumption and computational performance.

We would primarily implement the NeuCube on a software based system when principles of reconfigurability are at a premium, and the power cost is not a concern. Software based simulation of such a framework is ideal in the early stages of a model's development. See Chapter 6 for further discussion on this topic, and considerations based on the specific software simulation tool chosen.

5.5.2 Hardware Based Simulations

Generally speaking, a hardware based simulation will provide advantages in absolute scaling, power consumption, and performance guarantees. The actual degree to which they are advantageous in terms of power consumption is dependent on the specific hardware system. For the very lowest power cost implementation of the NeuCube we would look to use a system like the cxQuad neuromorphic VLSI ASICs of Indiveri, Corradi and Qiao (2015). This power advantage comes at the cost of model flexibility. In general, we would consider an implementation on neuromorphic hardware after we had implemented a reference version of the network in software, and shown it to be viable given the constraints of these platforms. See Chapter 7 for further discussion on this topic and considerations for some specific hardware systems.

5.5.3 Heterogeneous Computational Platforms

One example of an issue caused by functional differences in the computational platform used can be seen in some detail in Section 6.4.4. It is vital that future development of cross-platform models take such issues into account. A discussion of mitigation strategies for such issues can be seen in Chapters 6 and 7, where reference implementations of the NeuCube framework for different software and hardware platforms, and a generalised development framework for such implementations are introduced. See Section 7.4.3 for an example of the alterations necessary to a software-based implementation of the NeuCube for an implementation on the SpiNNaker neuromorphic hardware platform.

In effect, we must ensure that the NeuCube model we wish to implement will function adequately on the most-constrained platform; for example, when developing a NeuCube model for both software and neuromorphic VLSI, we must ensure that both systems can operate in a satisfactory manner using the particular synapse and neuron models physically implemented on that particular chip.

5.6 Design Methodology Overview

To codify this framework, then, we introduce the following methodology. It should be read in the context of this chapter, and indeed, the rest of this thesis. In order to optimally design a NeuCube-based SNN system for a given application, we must:

1. Define the *a-priori* information we have about the data.
2. Define our intended application.
3. Select a computational platform.
4. Select the most appropriate encoding scheme.
5. Construct a representative reservoir:
   (a) Define the inherent structure of the data (if any) and represent this in a 3D structure of neurons.
   (b) Define the most meaningful representation of the data collection context and use this to structure the input locations.
   (c) Generate a connectome, preferably based on knowledge of the data.
6. Create an output device based on our intended application and the data context.
7. Define whether the CNGM is necessary, and if so, what the initial point of the genes modelled should be.

We apply this framework with the theoretical support of the review in Chapter 3. Following this framework, it should be clear at each point what the optimal design decision is.

5.7 Chapter Summary and Conclusion

Here, a design methodology intended to codify and regulate the heuristic decisions necessary to design a NeuCube SNN system for arbitrary SSTD has been introduced. The development of NeuCube based systems is heavily informed by the specific data source we intend to use; it is from the structure and nature of this data that we draw the structure of the NeuCube reservoir. Additionally, other properties of the data such as its speed or density of channels inform our design decisions with regards to the system as a whole. In particular, these affect the encoding scheme chosen, and the parameters of the reservoir neurons and synapses. Our intended application informs the type and nature of the output device chosen.

A subsequent consideration is that of the computational platform selected. These systems have unique constraints depending on their underlying paradigm. Software based simulations are more flexible in terms of the network simulated and more

computationally costly, while in neuromorphic hardware systems we sacrifice some flexibility for higher computational efficiency and lower power consumption. Finally, this methodology has been summarised in a set of simple questions, the answers to which will inform the design decisions made.

To codify this system will introduce more rigour and repeatability to the process of creating NeuCube systems. At present this is very much a heuristic basis, with very little in the way of theoretical background for the majority of design decisions made. This is largely a property of the fact that this is a new model, and has yet to be explored in depth. Additionally, there is still not a rigorous information theoretic approach to the design of SNN systems in general. Some literature exists for the design of LSM systems, but this is not directly applicable to the NeuCube's dynamic structure as LSM use fixed, randomly initialised structures with no evolutionary functionality. There is some value in a more comprehensive assay of the LSM literature to identify any areas of potential overlap in future. This should especially assess any theoretical judgements in the area of readout functions in LSM, as there is the potential for this to be applied to optimise the linkage between the NeuCube reservoir and its readout functions. With the knowledge that the NeuCube conceptually draws inspiration from the specific data it is applied on, we have the opportunity to make better informed heuristic choices. It is the intention of this design methodology to make these choices more explicit and ensure that they are made in a meaningful way.

Contributions of this Chapter

1. Introduction of a design methodology for SNN within the NeuCube Framework.
2. Empirically established guidelines for design and development of
   (a) Encoding systems,
   (b) Reservoir design, especially the topology and connectome of these reservoirs,
   (c) Reservoir input location mappings,
   (d) Output devices such as classifiers or regressors, and
   (e) Application and inspiration of the CNGM.
3. Summary of considerations for SNN simulation on different computational platforms.
4. Identification of a possible method for the automated optimisation of the Threshold-Based Temporal Difference (TD) encoding algorithm.

Peer Reviewed Literature With Direct Contributions from this Chapter

1. Kasabov, N., **Scott, N. M.**, Tu, E., Marks, S., Sengupta, N., Capecci, E., Othman, M., Doborjeh, M., Murli, N., Hartono, R., Espinosa-Ramos, J.I., Zhou, L., Alvi, F., Wang, G., Taylor, D., Feigin, V., Gulyaev, S., Mahmoud, M., Hou, Z.-G. and Yang, J. (2016). Evolving Spatio-Temporal Data Machines Based on the NeuCube Neuromorphic Framework: Design Methodology and Selected Applications. *Neural Networks*. Special Issue on Learning in Big Data. Elsevier. doi:10.1016/j.neunet.2015.09.011

Chapter 6

Software Design Methodology and Implementations

> There are billions of neurons in our brains, but what are neurons? Just cells. The brain has no knowledge until connections are made between neurons. All that we know, all that we are, comes from the way our neurons are connected.
>
> — Tim Berners-Lee
> (*Weaving The Web*, 1999)

One of the primary aims of this thesis is to provide a general, platform-agnostic framework for the implementation of the NeuCube. Interestingly, despite the significant differences between concrete implementations of the NeuCube in software simulation or neuromorphic hardware, with careful consideration the core principles and design patterns behind a robust and efficient NeuCube can remain the same.

The considerations discussed in this chapter – with some obvious caveats based on the specific context in which it is developed – can be generalised to SNN systems other than the NeuCube. Obviously, it is not the implementation level details that we refer to here, but the high-level *design principles* (such as those discussed in Section 6.1.1).

6.1 A General Framework for Implementation of the NeuCube

In this chapter, we intend to introduce one primary software framework (Section 6.3) for implementation of the NeuCube. A secondary discussion will address a wholly-integrated form of the NeuCube, which has been termed the Core architecture

(Section 6.6). These approaches are developed within the Object-Oriented design paradigm. In addition, a tangible implementation of this first framework, written in Python using the PyNN library (Section 6.4) is discussed. These frameworks will be introduced in the context of the Design Philosophy (Section 6.1.1) which has informed the decisions preceeding them.

6.1.1 Design Philosophy

In order to develop a sufficiently modular and multipurpose framework for the NeuCube, we draw inspiration from the Unix design philosophy. This philosophy informs the majority of the design decisions made on this software system, particularly with regards to modularity of code. Kernighan and Pike (1984, pp vii) summarise the Unix design process:

> Even though the UNIX system introduces a number of innovative programs and techniques, no single program or idea makes it work well. Instead, what makes it effective is the approach to programming, a philosophy of using the computer. Although that philosophy can't be written down in a single sentence, at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many UNIX programs do quite trivial things in isolation, but, combined with other programs, become general and useful tools.

Here, we consider 'programs' to be individual components in the NeuCube framework such as input encoding or the 3D reservoir, or modules in the greater NeuCube software architecture (*cf.* Figure 6.1). In Scott et al. (2013), aspects of this design philosophy in the context of the NeuCube framework were discussed. Further core design rules intrinsic to the Unix design philosophy were codified in Raymond (2003), extending Kernighan and Pike's early work. The following enumerated sections are paraphrased from this text and contextualised in the NeuCube framework. A number have been consolidated together some of the features codified in Raymond (2003) are redundant or better explained in conjunction with other features, in this particular context.

1. *Modularity; Separation; Simplicity; Parsimony; Robustness; Diversity; Composition; Extensibility:* Build programs out of simple parts connected by well defined

interfaces. Simplify where possible, as this is easier to develop and more robust. Programs should be flexible and open. Programs should be designed to communicate easily with other programs. In essence, design for the future.

As the NeuCube architecture as a whole explicitly incorporates aspects of modularity, so too should the implementation reflect this. If developing a NeuCube implementation in an Object-Oriented paradigm, consideration should be paid to best practices in OO such as limiting class coupling and cohesion, and encouraging the use of polymorphism and inheritance, and appropriate design patterns.

Open and simple data formats should be used for file Input-Output (IO) and inter-module communication. For example, the use of AER for streaming data and JavaScript Object Notation (JSON) files for parameter IO is discussed later in this chapter.

2. *Clarity; Transparency:* Emphasise clarity of code and documentation rather than highly optimised and unreadable code. Design software for visibility and discoverability.

The system should be comprehensively documented, and preferably, written in a self-documenting manner. Complex operations should be simplified where possible, or at least explained fully. Such operations should make use of libraries where possible, as these provide a reduction of complexity for the developer and subsequent readers of the code.

Additionally, readability and clarity are implicitly improved in the particular system introduced in Section 6.4 due to the use of Python, which is somewhat self-documenting, especially when written in conjunction with established code standards like Python's PEP8[1].

3. *Least Surprise:* Build on top of the potential users' expected knowledge; user interfaces should do the least surprising thing.

The user should be able to interact with the system in an intuitive and logical manner; that is, method names and interfaces should be obvious and predictable in their effects on the system.

A canonical example of this is that the `+` operator in a calculator program should add two numbers together. In our case, we should reasonably expect that a method call like `Reservoir.generateStructure(neuron_locations.csv)` should load a CSV file containing neuron locations and use that to create the structure of the NeuCube

[1] https://www.python.org/dev/peps/pep-0008/

reservoir, or `Classifier.train(data)` should train the chosen classifier on the given data.

4. *Silence; Repair:* Only output concise and necessary information. When programs fail, they should fail in a manner that is obvious and easy to diagnose, or in other words "fail noisily".

Options should be provided for the verbosity of the program's output, depending on the context it is being run in. Long experiments (particularly those with optimisation or large numbers of experiment repetitions) should provide some report of progress. Failures should be reported immediately and with as much information about the source as possible, preferably including the computational platform's equivalent of a stack trace.

5. *Generation; Optimization; Economy:* Avoid manual optimisation; write abstract high-level programs that generate their own code or use libraries. Prototype software before polishing it; your code should work before you optimise it. Value developer time over machine time.

Here, we can make use of libraries such as PyNN and Scipy when using Python, and the software support provided for other computational substrates. This allows us to leverage the (generally) highly optimised code in these libraries and reduce both computational cost and development timelines. We also avoid premature optimisation, and develop a system from which we can make such improvements iteratively as the system demands it.

Such a design philosophy is applicable regardless of the implementation context – we can use these principles equally when developing a NeuCube implementation for the SpiNNaker as we can for an ASIC or a commodity PC. Adherence to this philosophy will encourage future development of the NeuCube to be performed in an efficient and sustainable manner, and allow for it to be iteratively improved in the future.

6.2 Overall Software Architecture

The software systems introduced here exist as a part of a larger software architecture. As the NeuCube framework has grown from a neuroinformatics classification tool, additional software modules have been added to the basic version. See Figure 6.1 for a visual overview of the general software ecosystem. Of particular note are the M2 and M3 modules; these, along with the M4 Visualisation module, have been introduced

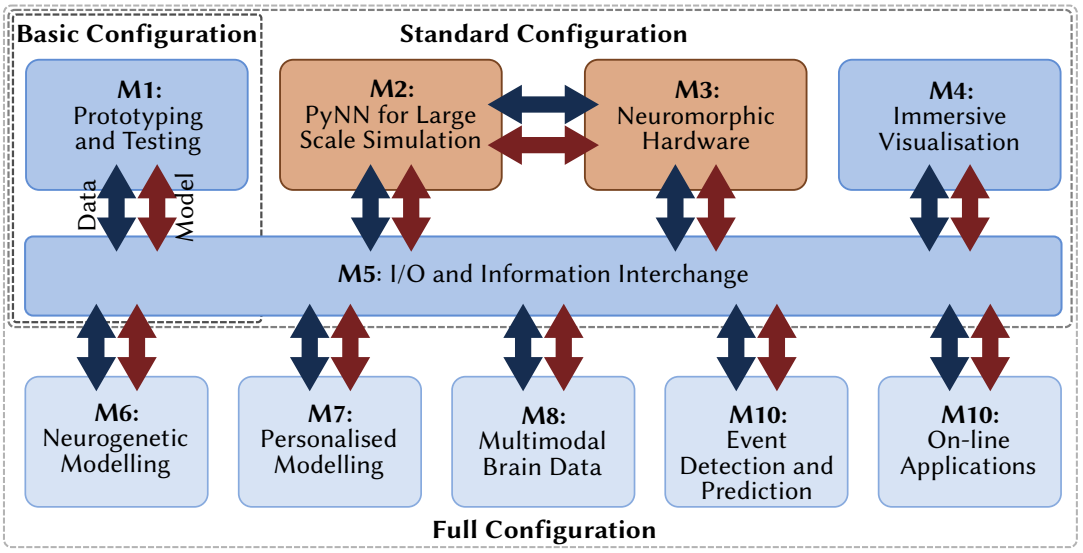


Figure 6.1: Block diagram of the NeuCube software architecture. Blue and red arrows represent data flow and saving/loading trained NeuCube models respectively. This thesis introduces implementations for Modules 2 & 3 (orange boxes), along with a general implementation framework for the remaining modules.

for the first time in this thesis. The other modules, save for the M1 Prototyping module, have commenced development only recently, and are not yet functional.

It is the intent of this chapter to introduce a generalisable software architecture for the implementation of these systems, and the refactoring of the M1 module. This chapter will also introduce an implementation of the M2 (software) module, and lead in to the M3 (neuromorphic hardware) module discussed in Chapter 7. Additionally, this will introduce a reference implementation of the NeuCube in Python, which is intended to be the basis of further development. This system has been developed with concepts of modularity and extensibility in mind, and is therefore a superior platform for the future of the NeuCube framework due to these factors.

Importantly, these modules must be identical in their shared functionality; for example, it must be expected that the calculation of the membrane potential of a LIF neuron must be identical across the modules, regardless of their extra features. At present this is not the case. See Section 6.4.4 for a discussion of this issue in some depth. This issue will be mitigated somewhat in future developments by the principles introduced in this thesis. Additionally, the future NeuCube Core system architecture discussed in Section 6.6 should mitigate this issue further, by providing a common set of functions that are interacted with by applications developers, rather than them implementing features from scratch.

6.3 A Reference Object-Oriented NeuCube Design

Aspects of this reference NeuCube design have been published in some detail in both Scott et al. (2013), and in a more general context in Kasabov et al. (2015). Here, some technical specifics relating to an object-oriented implementation of the NeuCube, including the design pattern, class structure, and internal communication schemes are introduced. A software architecture has been detailed in UML format in Figure 6.2. This section defines the software architecture used to develop the later systems in this thesis, which have been implemented in Python using the PyNN library in Section 6.4, and on the SpiNNaker device in Section 7.4.3. This design is not presently used elsewhere in the NeuCube; Modules 1, and 4–10 as defined in Figure 6.1 either use their own software architecture or are not yet under development. It is likely that this design is appropriate for use in the other identified NeuCube Modules, but it has not yet been adopted or extended for their use.

6.3.1 Software Design Pattern

With the design philosophy described in Section 6.1.1 in mind, it is logical at this point to discuss the software design pattern suggested for the reference implementation of the NeuCube.

A software design pattern is a general system towards a solution for a commonly occurring problem within a given context in software design. In this sense, they are simply best practices in software development, with the understanding that software development generally shares common challenges even across different fields. They are generally considered to be the intermediary layer between programming paradigms (*e.g.* Object-Oriented, Functional) and a specific algorithm or program.

The *de-facto* standard text in this area is Gamma, Helm, Johnson and Vlissides (1995). These authors are the so-called 'Gang of Four' in software architecture nomenclature. More recently, McConnell (2004) has provided an updated discussion of software design patterns. C. Zhang and Budgen (2012) show empirical evidence for the usefulness of design patterns in providing a framework for maintenance and extension – two areas of software development which this system is concerned.

The following software architectures are developed under the assumption that we will use the Template Method behavioural pattern. A behavioural pattern is a design pattern concerned with the activity of the system and how the software structure facilitates code reuse and communication.

Template Method Behavioural Pattern: The basic concept behind the Template Method design pattern is relatively simple. We create a (generally abstract) class representing the necessary steps for a general algorithm or operation. We then create a class that implements these steps with any necessary extension.

In this case, we will use the example of a classifier. No matter which specific algorithm (*e.g.* SVM, MLP, Naïve Bayes, deSNN, *etc.*) is used, all classifiers have a number of basic steps required in common; in particular, in a simplified case one must first *train*, and then *verify* the classier. We are therefore led to the basic template class:

```
abstract class GenericClassifier
  public abstract train()
  public abstract test()
```

In this case, these methods are abstract as they are not directly implemented here. We then wish to implement both an SVM and a deSNN. Obviously these are vastly different classification paradigms; SVM is an analytical technique, and the deSNN uses SNN and evolutionary computation principles. However, despite their differences, as long as they both implement the abstract methods defined in their template class (`GenericClassifier`) with the same input parameters and outputs, they can be swapped in-place with each other with no further code changes. Any extra methods required for the specific classifier is implemented in these subclasses. This design principle is used extensively in the development of the NeuCube and will be explored further in the following section.

Analagously, we could consider the Template class to be a 'motor vehicle' with subclasses such as 'bus', 'car', 'truck'. All of these share similar properties (an engine, wheels, passenger space) but implement them in different ways. Despite this, they can interface with the greater environment (roads) in the same way (steered with a steering wheel, passenger entry is through doors, *etc.*).

It would be a natural extension of this architecture to incorporate the Strategy Pattern. The Strategy pattern is conceptually similar to the Template pattern, except that the subclass (*e.g.* which classifier to choose) is selected automatically by the system. This technique could be implemented in the future when the optimisation strategies are better developed. We could utilise the strategy pattern in extending this system as an automated process for non-expert users. Consideration should be paid to this concept if commercialisation of the NeuCube is eventually sought and the user base is expanded to those inexperienced in SNN systems.

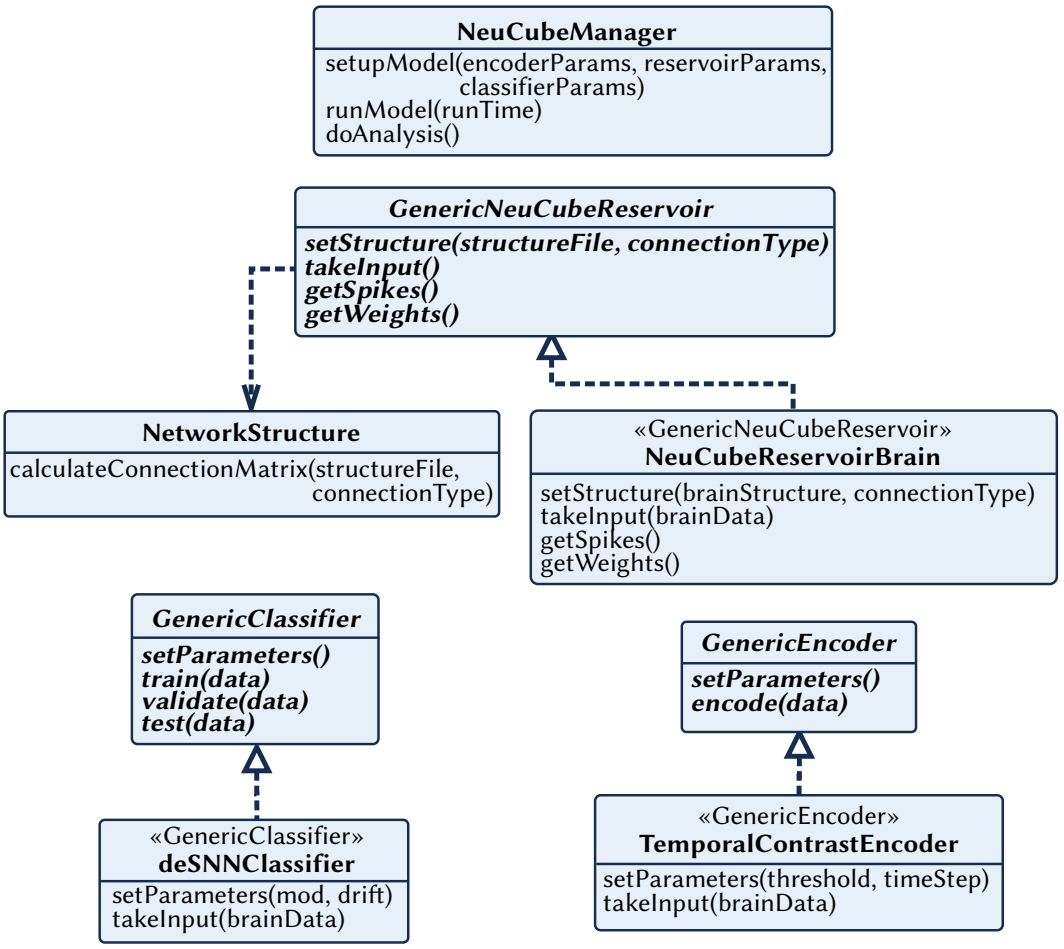


Figure 6.2: Simplified UML diagram of the reference Object-Oriented NeuCube implementation, showing only major methods of each class. These are example classes only, and the architecture shown in this figure should be implemented in the context of the design guidelines established in this chapter and with respect to the computational patform used (*e.g.* SpiNNaker, commodity PC). This design has been implemented in Python, and is discussed further in Sections 6.4 and 7.4

6.3.2 List of Classes

Here, a suggested list of classes for a basic implementation of the NeuCube is introduced, and visualised in Figure 6.2. This is relevant only for an *object-oriented* implementation of the NeuCube. A functional or imperative programming style implementation would likely be considerably different to this set of suggestions. By functional, we mean using a 'functional' language such as Lisp or Haskell. A functional approach to the NeuCube is not addressed here, although it would be an interesting exercise to implement it in this way. The NeuCube Core approach, an alternative implementation developed independently from an SNN simulation engine, is discussed in Section 6.6.

It is also important to note at this point that this is merely a suggested list of classes and methods, which may vary in reality depending on considerations for the computational platform and application context. Regardless, this provides a baseline design for the implementation of a NeuCube-based system in most general contexts.

The following classes have been detailed with simplified Java-style method signatures, despite the fact that later in this thesis (*cf.* Sections 6.4 and 7.4) they have been implemented in Python. This is to ensure that the method visibility and inheritance structure is made explicit to the reader, as these are poorly delineated in native Python. In fact, the actual implementation language is arbitrary. The NeuCube could as easily be implemented in C/C++, Java, Python, MATLAB, or any language – as, in fact, it has been. Java-style syntax is used here simply for clarity.

A number of assumptions have been made to simplify this description. In particular, the calculation of neuronal and synaptic dynamics is assumed to be a solved problem; *i.e.* this is a higher level implementation of the general NeuCube framework on some computational platform, rather than a to-bare-metal implementation. With this in mind, the following assumptions are made in the subsequent sections:

1. A suitable SNN simulation backend (PyNEST, Brian, SpiNNaker, *etc.*) has been selected.
2. Utilities, such as those responsible for file loading and plotting functions, are omitted.
3. The development is performed in the context of the design guidelines introduced in Chapter 5 and this chapter, and the considerations for the specific computational platform.

The relative brevity of the code template introduced here and the implementation in Section 6.4 are made possible by this first assumption. Utilities have been ignored here as they are platform & case dependent, and would in any case be subsumed by other methods. For example, loading a Comma-Separated Value (CSV) file would be handled implicitly by the method to set up a reservoir structure. Additionally, only public methods have been identified here. Private or protected methods would be implemented on a case-by-case basis, and are in any case subsumed into the greater logic of the public methods.

6.3.2.1 Control Class

```
class NeuCubeManager
  public setupModel
```

```
  public runModel
  public doOutput
  public doAnalysis
```

This should be considered the 'main' class, containing the main logic loop of the system. This class is responsible for scheduling and initiating the simulation, and managing any ancilary steps required, including any automated plotting or statistical analysis.

`setupModel`

Subsumes the methods required for loading configuration files, encoding input data, and setting up the network strucuture. Prepares the system for simulation. Also sets up the chosen output device.

`runModel`

Runs the simulation, and manages any concurrent tasks such as streaming IO (*cf.* Section 7.4.2.7) or external influences on the network. Handles cleanup after the simulation is run, including readback of spike time records for neurons in the network, and synaptic weight matrices.

`doOutput`

Train and test the output device (classifier, regressor). If online output (*e.g.* for robotics applications) is required this method is subsumed into `runModel`. This method can be modified and separated into training and testing methods, to allow the system to be used repeatedly instead of 'one-shot' training.

`doAnalysis`

Perform any necessary automated analysis steps required, including statistical measures of spiking times and synaptic weights, and any plotting necessary. This method may be run interactively, or 'silently' when used in batch job form.

6.3.2.2 NeuCube Reservoirs

```
abstract class GenericNeuCubeReservoir
  public abstract setStructure
  public abstract takeInput
  public getSpikes
  public getWeights
```

These methods deal with the creation and management of the 3D NeuCube reservoirs, including the actual setup of the structure. We have not mentioned the calculation of network dynamics (*i.e.* the calculation of neuron spike times), as in this discussion this has been abstracted away to the computational platform. In reality, this function would most likely be delegated through this class. In the case we chose not to use a simulation platform, the calculation of such dynamics would be managed here and implemented in a new class.

This class is primarily abstract, or could be implemented as an interface. It has been abstracted to allow different NeuCube reservoirs to be implemented later with no changes to the interface code. In this way, NeuCube reservoirs using alternative dynamics such as the CNGM can be transparently implemented, as long as they all implement the methods defined in this class. This supports the Template Method design pattern identified in Section 6.3.1, and will allow for later implementation of the Strategy pattern where the selection of subclasses is automated.

`setStructure`

Sets up the 3D reservoir structure, given some file or data structure containing these locations. Generates the connectome from a formula (*e.g.* the distance-dependent probability in Section 6.4.3.1), some *a-priori* information about the data collection context such as DTI (*cf.* Section 2.1.1.2), or a given list of connections.

`takeInput`

Manage the input of data into the system. This is either some data structure with spike times, or streaming spike IO as discussed in Section 7.4.2.7. This data structure will be dependent on the implementation language; for example, in Python this would likely be a list or dictionary, whereas in Java it may be an ArrayList or similar.

`getSpikes`

After a simulation, return spike trains of neurons in the reservoir, whether this is the whole network or a given selection of the total neurons. In the case of streaming spike IO, this method will output these spikes in as close to real-time as possible, else they will be returned as a data structure containing key-value pairs of neuron-spiketimes (*i.e.* in the format $n_1 = \{t_1, t_2 \ldots, t_n\}$). This method is not abstract as the general implementation will be identical among all of its subclasses, as spike retrieval should not change significantly between different network structures.

`getWeights`

Retrieve the synaptic weight matrix of all or a subset of the connections in the reservoir after a simulation. Updated synaptic weights could be retrieved in real-time as they change, as long as the simulation overhead was acceptable. This method is not abstract as synaptic weight retrieval should be general among all subclasses.

```
class NeuCubeReservoirBrain extends GenericNeuCubeReservoir
  public setStructure
  public takeInput
  public getSpikes
  public getWeights
```

This class would implement a form of the `GenericNeuCubeReservoir` defined above, for the specific case of a NeuCube reservoir for brain data. In this, it would implement the defined methods in the context of that brain data; *i.e.* the reservoir would be structured using the Talairach or MNI brain data atlases, the connectome would be generated in a small-world manner or by using some DTI information, and so on.

Similarly, for the example of seismic data, a `NeuCubeReservoirSeismic` could be implemented, where the network structure and behaviour would be significantly different to a NeuCube reservoir designed for brain data.

```
takeInput
```

This method should accept encoded input spike trains in the particular format of the brain data collection device used, and pass it to the simulation. This may stream in online in the case of a BCI or Motor Neuron Prosthetic (MNP) control application (*cf.* Section 7.1.5.2), or it may be batched.

```
getSpikes
getWeights
```

In the case of the `NeuCubeReservoirBrain`, these methods can be omitted, as the general versions in `GenericNeuCubeReservoir` should suffice. In the case that some specific behaviour is required here, these methods can be implemented to override the generic version in the superclass.

6.3.2.3 Network Structure

```
class NetworkStructure
  public loadNetworkFile
  public loadNetworkMatrix
  public calculateConnectionMatrix
```

This class is responsible for the creation of the 3D structures of a NeuCube reservoir. It should be general enough that different structures can be generated without code changes, depending on the location file or data structure given as input. In the case of different connectome generation schemes (*e.g.* one based on DTI or a different small-world scheme) this class could be extended with a subclass that overrides this method.

`loadNetworkFile`

Load a file containing the 3D locations of the reservoir neurons, and any ancillary data required for the network structure setup.

`loadNetworkMatrix`

Load a file containing a known network structure; for example, one that has been pre-trained in a different module of the NeuCube. This should contain neuron positions, and a connection weight matrix for the known connections. If no known network matrix is available, use the `calculateConnectionMatrix` method below.

`calculateConnectionMatrix`

Calculate the connection matrix (*i.e.* the synapses between neurons) based on the loaded network structure, and some established formula for connection structure. The basic form used in the NeuCube is the distance dependent method established in Section 6.4.3.1. This method implements that part of the `GenericNeuCubeReservoir` unless it is overridden in a subclass.

6.3.2.4 Classifiers

```
abstract class GenericClassifier
  public abstract setParameters
  public abstract train
  public abstract validate
  public abstract test
```

This is the generic superclass of the classifiers or regressors (*i.e.* the output devices of the NeuCube). In this case, the output device is considered to be a classifier, as this is the default in the NeuCube model. It would be trivial to refactor this design to have a `GenericOutputDevice` which both the `GenericClassifier` and a new `GenericRegressor` would extend further. This is, in fact, the correct behaviour if we approach this design from an object-oriented point of view, but as no regressor is

yet planned in the NeuCube, we will follow the original approach here. The concepts introduced here would in any case, extend to this alternative design.

The concept behind making this class abstract is primarily that classifiers implementing these abstract methods can then be interchanged without modification of the code interfacing with them, in the same manner as the `GenericNeuCubeReservoir`. This facilitates later adoption of the Strategy design pattern, and adheres to the current Template Method pattern.

`setParameters`

Load and setup the required parameters for the classifier. This method is entirely dependent on the specific classifier selected.

`train`

Train or generate a classifier model for the given data using the pre-loaded parameters. The implementation of this method is entirely specific to the chosen classifier. For example, in the case of an SVM, in this method we would analytically create the separation hyperplanes as detailed in Cortes and Vapnik (1995). In the case of the deSNN, this will be discussed specifically in the next example. This process can be performed on-line in the case of some classifiers.

`validate`

Validate the pre-trained classifier model. This process is performed to validate the performance of the classifier with a given set of parameters against a known result set, and can be used in automated optimisation of the same. This process may continue the classifier's training if that is consistent with the algorithm selected (*e.g.* the deSNN can continue to evolve during validation but an SVM cannot). Validation may be performed on-line if this is supported by the classifier algorithm and the reservoir defined.

`test`

Test the pre-trained classifier model on an unseen data set, to provide independent classification performance metrics. This process may continue the classifier's training if that is consistent with the algorithm selected. Testing may be performed on-line if this is supported by the classifier algorithm and the reservoir defined.

```
class DeSNNClassifier extends GenericClassifier
  public setParameters
```

```
  public train
  public test
```

This is an example of an implementation of the `GenericClassifier` defined above, for the specific case of a deSNN (*cf.* Section 3.6.3.2) classifier.

```
setParameters
```

Loads and sets the deSNN-specific parameters, including the Mod, C, and drift parameters.

```
train
```

Inital training of the deSNN. Propagates the spike trains recorded from the NeuCube and generates the neuron clusters and synaptic matrices responsible for classification of those spike trains. In the case that the data is streaming, this training phase can be completed on pre-recorded data in the same way, or streamed in.

```
test
```

Validation and operational method of the classifier. In a normal case the training aspect of the learning algorithm would be switched off here, although in the case of the deSNN, the learning and adaptation is life-long. A distinction is made here for statistical purposes, to ensure that the accuracy is properly recorded. In the case that the input data to the greater NeuCube is to be streamed, this method could be run in parallel with the reservoir, and spikes transferred in real-time from the reservoir to this method.

6.3.2.5 Encoders

```
abstract class GenericEncoder
  public abstract setParameters
  public abstract encode
```

This is the generic superclass of all spike encoding methods implemented in the NeuCube. The ability to easily change encoding schemes is important in the NeuCube framework, as they can have significantly different dynamics and should be implemented in different scenarios. Creating a `GenericEncoder` will allow us to change encoders without alteration to the rest of the NeuCube software, and is consistent with both of the design patterns previously identified in this chapter.

```
setParameters
```

Loads and sets the parameters for the encoder. This method is specific to the encoding scheme selected.

`encode`

Performs the actual data → spike train conversion. In the case of on-line data streaming, this method must be able to support on-the-fly encoding of data.

```
class TemporalContrastEncoder extends GenericEncoder
  public setParameters
  public encode
```

This is an example of the `GenericEncoder`, where we implement a Threshold-Based Temporal Difference (TD) encoding scheme (*cf.* Section 3.5.1).

`setParameters`

Loads and sets the TD-specific parameters, including the spike generation thresholds, step size, and whether to encode both excitatory and inhibitory spikes or a choice of one.

`encode`

Encodes the given data into trains of spikes using the defined algorithm and parameters. Passes these to the NeuCube reservoir modules.

6.3.3 Inter-Module Communication

Each of these separate components and modules must be able to communicate with the others in some standardised way. Internal communication between components of a module is generally straightforward; as these will likely be within the same software package, provided that they follow the expected parameter format of the various methods called, their communication is subsumed by the application itself. More complex is the communication between the greater software modules, and with external devices like the DVS or an EEG collection device.

To address this communication, two primary forms of inter-module IO are proposed; the AER format for streaming or real-time data, and a JSON-based file format for the interchange of static or pre-prepared data.

6.3.3.1 Address-Event Representation

Address-Event Representation (AER) is a conceptually simple technique for the labelling of spike trains. In essence, it is a pairing of a neuron number or spatial location ('address') with the time that neuron spiked ('event'). This technique is typically used in neuromorphic hardware devices, including the SpiNNaker (Furber, 2012) and Institute for Neuroinformatics (INI) VLSI ASICs such as those introduced in Indiveri et al. (2010), Mitra, Fusi and Indiveri (2009) or Moradi and Indiveri (2013). Devices such as these may extend AER depending on their structural and functional requirements – for example, SpiNNaker encodes its 'addresses' as a chip-core-neuron heirarchy, to aid in its internal routing process (*cf.* Section 7.4.1).

This principle is perhaps best seen in the Dynamic Vision Sensor (DVS), a spike-based computer vision system detailed in Section 7.1.5.1. Here AER is used to represent a pixel location ('address') and its spike time ('event'). For a visual intuition of how this system works, see Figure 7.5. See Section 3.5.1 for further details.

At the 2015 Capo Caccia Cognitive Neuromorphic Engineering Workshop, a group developed a draft protocol for the general implementation of AER on heterogeneous systems. This protocol is discussed in draft form by the group participants in Rast et al. (2015). This work was a response to the issue of intersystem communication in neuromorphic systems. As these hardware devices become more prevalent, it becomes more important that they are able to communicate with one another. However, there is as yet no universally agreed-upon standard for the specific format of an AER 'packet', nor any attempt to make it suitable for transmission over industry-standard networks like Ethernet. This specification document addresses the format used in the interchange of such messages between compatible AER-generating and -receiving devices, providing a common message interchange medium. It is at present a draft specification, and therefore we will not go into significant detail here, as it is subject to change at any time.

As this protocol can be used over a wide variety of communication means, it does not specify the transmission layer; rather, it focuses only on the content of the transmission. It is expected that the interface can be bidirectional, *i.e.* that a device can both issue and receive spikes. However, it is not required that a given device must be able to do both. A device could be either a blind issuer of spikes or a passive receiver.

This protocol was developed with all major academic neuromorphic systems developers (*e.g.* INI's ASICs, Heidelberg's FACETS-BrainScaleS, SpiNNaker) and it is therefore likely that future devices from these groups will work towards compatability with this protocol. NeuCube systems should in future implement this protocol for compatability with such hardware devices, and any software systems also supporting it. Section 7.4.2.7 discusses an example of how spike IO can be implemented with AER when running the NeuCube on a SpiNNaker device.

In the case of the NeuCube, the AER paradigm would be best used for streaming data, rather than parameter values or network structures. Here we would first load a network structure either programmatically or via the JSON-style format discussed in the next section, and then stream the encoded input data in via AER. This would allow the NeuCube to be applied in real-time contexts like robotics or prosthetics control more easily than with file-based IO of spike times.

6.3.3.2 File-Based IO with JSON

JavaScript Object Notation (JSON) is a general specification for text-based file interchange. As the name suggests, it was initially designed for use with Javascript, but has been adopted as a more human-readable and concise alternative to XML. In our case, the JSON format has been chosen to ensure that in the greater NeuCube software architecture, we have true cross-platform support. Bearing in mind that various modules of the NeuCube are currently written in three different languages (MATLAB, Python, or Java), it is difficult to implement a binary format which can be easily interpreted by all of these languages. It is, of course, possible, but it would require us to implement a custom binary format. Coupled with the concept that these systems may eventually be remotely accessible over a network, and that they are currently implemented on both Windows- and Linux-based platforms, the choice was made to use a flat file format for data interchange. This is also consistent with the design philosophy in Section 6.1.1, where modularity, simplicity, and transparency were emphasised in making implementation decisions. In these files, values are encoded as key-value pairs, with a syntax very similar to Python. It is a trivial matter to map this file structure into our language of choice.

Unfortunately, after the decision to use a JSON-format for important file-based IO was made, these files were implemented without group consultation by another developer, in the MATLAB M1 module (introduced in Section 6.4.4). An example of the files generated is too long to print here. A significantly reduced version is

available online.[2] Presently, a JSON representation of the typical brain-shaped 1471 neuron reservoir structure is around 15 MB, when in theory it should be less than $\approx$ 200 KB. Detail will not be entered into about the issues with this representation, as an attempt is currently being made to bring this back in line with the principles established in this thesis. In particular, a sparse representation of the connectome will be implemented, as this is by nature a sparse matrix. Representing this connectome as a dense array will result in exponential growth of the representation, a behaviour which is untenable as we move to larger network sizes. Representing it as a sparse matrix will significantly save on file size, as at present, the majority of the data in this file is redundant or otherwise unnecessary.

All that is necessary to represent in this file is the neuron locations in 3D space, their basic parameters, the connection matrix, and the weight matrix. The first must be a dense $n \times 3$ array, and the others sparse $n \times n$ matrices, where n is the number of neurons. Neuron parameters can be consolidated together, as at present we model these neurons homogenously; *i.e.*, their dynamics such as spiking thresholds or leak rates are identical. Currently, the spiking history of the reservoir is saved here, along with a number of redundant data structures (*e.g.* the input location names, which should be contained in the input data file), but this should be a separate file as it can change independently of the structure.

Of course, this type of representation will begin to face limits when the network size increases. This is an issue which we have yet to face in our development of the NeuCube. However, in the near future, we may need to seek an alternative formulation for connectivity matrix definition, and so on. One possible alternative to this list-based network structure definition is the Connection Set Algebra introduced in Djurfeldt (2012) and implemented in a number of simulation platforms. This can represent a network structure deterministically as a set of formulas, which can be interpreted in the simulator. This representation is advantageous here as it would be constant regardless of network size. but can still be deterministic with regard to the networks it produces.

A simplified JSON-formatted configuration file, as used in this version of the NeuCube, is given in Appendix F. Here, the file links to the other files required – neuron positions, input data, and input positions. This is a more efficient and compact representation of the necessary configuration data than the previous configuration system employed in M1.

[2] https://gist.github.com/nmscott/e569c2b957849242ce67

6.4 Implementation of this Framework Using PyNN

A concrete implementation of the NeuCube developed using the principles discussed in this chapter is introduced here. It is hoped that this system will form the basis of a 'reference' NeuCube; *i.e.* a system used for benchmarking new implementations of the NeuCube, and one developed in a mindful way.

The general application flow has been discussed in Section 6.4.2. However, the primary contribution of this section is not given here; instead, it is the full listing of the code required to realise the NeuCube in PyNN, given in Appendix C. Key segments of the code have been reproduced and explained in this section. Certain challenges were encountered in the development of this implementation, and are primarily discussed in Sections 6.4.3 and 6.4.4.

6.4.1 Why PyNN?

The motivation to use PyNN in developing the primary code artefact of this thesis should be clear at this point, due to the nature of the system being developed and the nature of PyNN. However, to make explicit for completeness' sake, the reasons are straightforward:

- PyNN allows for a more concise model representation than developing for a simulation tool directly.
- PyNN provides cross-simulator support – a much greater number of computation platforms are supported by a single PyNN application, when compared to developing for one of these computation platforms specifically. In addition, with minimal changes a PyNN application can also be run on neuromorphic hardware like the SpiNNaker.

PyNN is therefore a natural choice for a system such as the NeuCube, which is intended to run across multiple platforms (including hardware) and be modularly configurable. For further clarification of PyNN and why it is an optimal choice for the NeuCube, see Section 3.8.3.

6.4.2 Program Overview

The simulator selection in PyNN is based on which PyNN package we import. For example, to use the NEST or Brian simulators, we would import PyNN as

`import pyNN.nest` or `import pyNN.brian`, respectively. Similarly, to implement the SpiNNaker port of PyNN known as sPyNNaker, we can either `import sPyNNaker` or `import pyNN.spinnaker` to keep consistency with the rest of the PyNN implementation. The SpiNNaker will not be discussed here, as the modifications necessary to convert our PyNN implementation to sPyNNaker are covered in some detail in Section 7.4.3.

An application written in PyNN requires at minimum, three method calls:

1. `setup()` which initialises the simulator platform and sets any simulator-specific parameters such as the time resolution;
2. `run(time)` which intialises and runs the actual simulation for `time`; and
3. `end()`, which 'cleans up' after the simulation is completed, freeing memory and so on.

For the NeuCube implementation, these methods have been implemented separately in the NeuCube reservoir and the SNN classifiers, as these are separate networks and as such cannot be simulated together.

In reality, a number of other methods are required to ensure that the network and simulation are set up and run succesfully. Here we briefly introduce the application flow for implementing the NeuCube in PyNN. In cases where the code is complex or otherwise non-obvious, the code will be reproduced when explained. Key areas of code which merit intensive discussion are given in Section 6.4.3. Note that this implementation generally follows the software architecture introduced in Section 6.3.2.

The basic application flow for a single operation of the NeuCube is as follows:

1. Load configuration files

A configuration file (or more precisely, files) will contain the necessary network parameters for a NeuCube model; for encoding (the scheme chosen, scheme parameters, whether to stream or batch this encoding), for the network (neuron type chosen, neuron parameters, network structure, connectome type), and for the output device (classification or regression scheme chosen, parameters for the previous). One such example configuration file is given in Appendix F.

2. Encode data with the chosen encoding scheme

Firstly, the selected data is loaded, and the encoding scheme set up. In the case of the TD encoder, we set the spike emission threshold, timestep, and whether to encode

inhibitory spikes. The data is then propagated through the encoding scheme, and as streaming IO is not yet implemented, it is returned as a list of lists.

The encoding step could be implemented in parallel with the actual simulation; encoding only the information which is required at that point in the simulation. This would reduce the memory usage of the application when large (*e.g.* for dense channel or temporally long) data is processed in the NeuCube.

To this point, we are not yet using PyNN, or any simulation backend. Our encoding is implemented in standard Python; in some cases it may be accelerated using libraries such as SciPy or NumPy which provide optimised mathematical operations.

3. Set up reservoir

Prior to setting up the network, we must select our simulation platform and import it. As discussed in Section 7.4.3.3 this import statement will differ depending on our simulation platform. In this case, we will use NEST, although it is equally possible for use to use Brian or the SpiNNaker in its place. To import our simulator, we simply `import pyNN.nest as p`. This aliases `p` to PyNN, allowing us to shorthand the later method calls and keep them distinct from non-library code.

We then need to set up the structured NeuCube reservoir. We firstly set up the neuron parameters and structure. The parameters of the neurons are easily set; in PyNN, the syntax to create a Python dictionary to contain the key-value pairs for the parameters is straightforward:

```
cell_params_lif = {'cm':0.25, 'i_offset':0.0, 'tau_m':10.0,
                   'tau_refrac':2.0, 'tau_syn_E':3.0, 'tau_syn_I':3.0,
                   'v_reset':-65.0, 'v_rest':-65.0, 'v_thresh':-50.0}
```

Listing 6.1: Definition of the NeuCube's neuron parameters in PyNN.

We then call the network structure generation method, which has been implemented as discussed in Section 6.4.3.1. This generates two primary lists, respectively representing the excitatory and inhibitory connectomes. Subsequent to generating the network structure, we select our input neurons using the method discussed in Section 6.4.3.3. As these methods are all based in native Python, we have not yet needed to access PyNN's methods. Were we to use the inbuilt network structure generation methods of PyNN, we would need to call PyNN's `setup` method prior to generating the network.

Next, we set up a `data_prefix` to prepend the saving of plots and result files from the current simulation. This prefix is automatically generated based on the simulation

start time, run (simulation repetition) number, and data sample indentifier. In this way, we can automatically tag and save data in a traceable manner for large simulations.

At this point, we must call PyNN's `setup()` method to initalise both PyNN and its interface to our chosen simulation system. Here, we call `setup` with the parameters `timestep = 1.0` and `min_delay = 1.0`, to ensure that all simulations run have an identical time resolution. This is also the minimum time resolution and axonal delay possible for the SpiNNaker device, ensuring that our simulation behaviour should be cross-compatible between software and SpiNNaker simulation.

From here, the methods of PyNN are made available to us. Firstly, we initialise a population of neurons to represent our reservoir. This is the total network size. Here, we use the `cell_params_lif` previously shown in Listing 6.1 to define the neuron properties.

We then set up the parameters of the chosen STDP model. In the case of the NeuCube, we use a non-homogenous form of STDP, wherein the excitatory and inhibitory populations have different parameters to ensure that the weight of the inhibitory synapses is never such that the network is silenced.

```
# Set up excitatory STDP
timing_rule_ex = p.SpikePairRule(tau_plus=20.0, tau_minus=20.0)
weight_rule_ex = p.AdditiveWeightDependence(w_min=0.1, w_max=1.0,
                                            A_plus=0.02, A_minus=0.02)
stdp_model_ex = p.STDPMechanism(timing_dependence=timing_rule_ex,
                                weight_dependence=weight_rule_ex)
# Set up inhibitory STDP
timing_rule_inh = p.SpikePairRule(tau_plus=20.0, tau_minus=20.0)
weight_rule_inh = p.AdditiveWeightDependence(w_min=0.0, w_max=0.6,
                                            A_plus=0.02, A_minus=0.02)
stdp_model_inh = p.STDPMechanism(timing_dependence=timing_rule_inh,
                                 weight_dependence=weight_rule_inh)
```

Listing 6.2: Definition of the NeuCube's Spike-Timing Dependent Plasticity rules in PyNN.

The inputs are then set up. We use the encoded spike times in the lists created earlier, and feed one channel each to a number of `SpikeSourceArray` spike generators, which are attached to our selected input neurons.

The excitatory and inhibitory populations are then connected together. This step (Listing 6.3) is somewhat convoluted at first glance. To connect the excitatory population to the whole population of neurons (including inhibitory ones), we use the `excitatory_connector` list generated earlier. We then repeat the process for the

inhibitory neurons with the `inhibitory_connector`. The difficulty in this operation is discussed in Section 6.4.3.1.

```
connected_excitatory_neurons = p.Projection(neurons, neurons,
            excitatory_connector,
            synapse_dynamics=p.SynapseDynamics(slow=stdp_model_ex),
            target="excitatory")
connected_inhibitory_neurons = p.Projection(neurons, neurons,
            inhibitory_connector,
            synapse_dynamics=p.SynapseDynamics(slow=stdp_model_inh),
            target="inhibitory")
```

Listing 6.3: Connection of the NeuCube's reservoir network in PyNN.

Data recorders are then attached to the neuron population. These record spike timings for all of the neurons in the population, and can also record other properties such as membrane voltage over time.

At this point, the network is set up, and ready to be loaded into the computational platform for the actual simulation to be run.

4. Run reservoir simulation

Once the network has been set up, it is a relatively trivial exercise to actually run the network. We simply call `p.run(sim_time)`, where `sim_time` is the length of the simulation in milliseconds – *i.e.* the length of each input sample. Internally, this method loads the network into the simulation platform and executes the actual simulation (the calculation of spike dynamics over time).

At the end of a simulation, our data recorders will contain the spiking history of the reservoir, and this can be saved. Additionally, the synaptic weight matrix will have evolved. This too, can be saved if desired.

We then `p.reset()` the network. This method resets the network to an initial state; the internal clock of the network is reset to zero, the neuron membrane potentials are set back to their resting state, and any spikes in transmission are deleted. The weight matrix and network structure are, however, retained in memory. Next, we reset the input spikes to the next pattern, and `run` the network again. By repeating these steps, we can perform the unsupervised learning process in the reservoir, as the weight matrix will be iteratively updated with each sample presented.

This process is simple only for software simulation; in the case of the SpiNNaker device, the unsupervised learning process is much more involved as the `reset` method is not supported. This issue is discussed in Section 7.4.3.2. After the training

or validation process is completed, we call `p.end()` to clean up and free the resources held by the simulator.

5. Set up output device

The steps discussed in the next two sections will be more general; their implementation is heavily dependent on the type of output device chosen. There are, however, some processes in common among all of the possible output devices. From here, we will use the example of a deSNN classifier, identifying the key steps necessary in implementing that particular learning algorithm in PyNN and in this context.

This will involve mapping the parameters given in the configuration file to the device; in the case of a deSNN, this will be the Mod, C, and drift parameters.

The rest of the set up follows the same basic process as that given for the reservoir. However, instead of generating a large population of neurons, we begin with a single neuron, with n input neurons (where n is the number of neurons in the reservoir). This is due to the fact that in PyNN, dynamic generation of neurons is not supported; *i.e.* we cannot evolve new neurons on-demand as is required in the deSNN algorithm. Instead, these new neurons will need to be added to the population in a process explained in the next section, similar to the manner in which the original population is generated.

6. Run output device

To actually run and evolve a deSNN classifier in this context is less trivial.

Effectively, we will add one neuron every time a sample is propagated through the classifier. Recall from Section 3.6.3.2 the basic algorithm. In this learning method, a new neuron is evolved for each sample, and compared against the current neurons of that class. If they are sufficiently similar, they are merged. Logically then, we begin the network with one neuron to represent the first sample. We perform the deSNN learning algorithm, including the calculation of the PSP and PSP_{max}.

The synaptic weights are saved, and `end()` called. For the next training sample, we reconstruct the original network, with an extra output neuron. The first neuron's synaptic weights are restored from our saved version, and the new neuron's synaptic weights are generated according to the deSNN algorithm.

The repetition of this process is somewhat inefficient, but as there is no explicit ability to evolve new neurons in the same architecture, we must work around this limitation.

It would also be possible to create a full network of neurons (*i.e.* one neuron per sample) and simply set the as-yet unused synaptic connections to a weight of 0. However this is an inefficient method, as we are then incurring an unnecessary computational cost for the reserve neurons – their dynamics are calculated at each timestep regardless of whether incoming spikes occur.

Clearly from these last two steps, the implementation of an output device is very much contingent on the method chosen; however, this example should identify a few of the key challenges in implementing such an algorithm in this context.

In the following sections, some of these points will be addressed in more detail.

6.4.3 Key Code Sections Explained

As mentioned in Section 6.4, key sections of the PyNN code worth deeper explanation will be discussed here. If necessary for context, these will be reproduced; otherwise, for a listing of the code required to run the NeuCube in PyNN[3], see Appendix C.

6.4.3.1 Manual 3D Structure Generation

Initially, the 3D structure of the NeuCube was developed using the native PyNN `Structure` classes. This provides a concise and efficient representatation of 3D space in a simulation, and allows access to the native PyNN distance calculation and connection generation methods. To implement custom network shapes in PyNN, it is possible to extend the `space` package's `BaseStructure`. We extended this class to create `NeuCubeStructure`, which is a generic class designed to load CSV files containing arbitrary neuronal locations and build a 3D structure from these. See Appendix C.8 for the Listing of this deprecated method of structure generation.

However, as development continued, and the structures became more complex, the implementation PyNN provides were shown to be inadequate. Primarily, these structures were not able to support two populations of neurons within one structure. This means that the native PyNN structure was unable to represent both excitatory and inhibitory populations within the same list of 3D neuron locations. To mitigate this issue, there are two primary options:

1. Generate separate lists for excitatory and inhibitory populations with the correct proportions

[3] Also found online at https://github.com/nmscott/NeuCube_PyNN

2. Manually implement methods for connection structure, and distance generation methods.

The simplest option is to split the current neuron location list into two separate lists – one excitatory and one inhibitory – which proportionally occupy the same space. However, this is not an effective solution due to a further limitation of PyNN.

As mentioned in Section 3.8.3, PyNN provides a number of connection schemes, including all-to-all, fixed-probability, and from-file connectors. Perhaps most usefully for us, it also provides a `DistanceDependentProbabilityConnector`, which takes an arbitrary function $P(c_{ij}) = F(d_{ij})$ where $P(c_{ij})$ is the probability of a connection between neurons i and j and $F(d_{ij})$ is some function of the distance d between neurons i and j. By 'distance dependent connection probability' it is meant that $P(c_{ij}) \propto e^{-d_{ij}}$, *i.e.* that the likelihood of a connection decreases as inter-neuron distance increases.

These connection schemes operate *within* neuronal populations, and are not applicable when we have two populations sharing the same space. Typically, this would not be an issue, as these populations can generally be separated with no loss of functionality. Unfortunately, as the 3D structure of the NeuCube reservoir is meaningful, we cannot separate the excitatory and inhibitory populations in a structural way.

Therefore, a 'manual' (*i.e.* not native PyNN) connection generation method was implemented. See Listing 6.4 for the pseudocode of this implementation. This operates across populations of neurons in the same space. At present, it is written to calculate distance-dependent connection probabilities both within and between two populations of neurons, although this could easily be generalised to the multiple populations of neurons required for a CNGM or for different connection schemes.

This implementation of an exponentially inverse distance dependent connection probability $P(c)$ between neurons i and j is straightforward, and supported by Gerstner, Kistler, Naud and Paninski (2014), though in their case autapses are not considered:

$$P(c_{ij}) = \begin{cases} p_s \, e^{-d_{ij}} & \text{if } d_{ij} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

Where p_s is a probability scaling factor and d_{ij} in this implementation is the Standardised Euclidean distance between neurons i and j. If p_s is zero, no connections will be made, and in the case that $p_s = 1$ the network will be fully connected. The case test is required to prevent self-referential connections (autapses). If the distance between neurons is zero, it can be reasonably assumed that they are the same neuron, and

therefore $P(c_{ij})$ should also be zero, unless autapses are acceptable. Removal of this check would allow autapses if they are required in the model. It would be trivial to extend this formulation in the case that different connection rules were required either between or within neuronal populations. A pseudocode implementation of the process is given below.

```
for (presynaptic neuron i = 1 to size(input))
  for (postsynaptic neuron j = 1 to size(output))
    calculate distance_ij
    if distance > 0
      calculate p_connection_ij by Equation 6.1
      generate random number r ∈ (0,1)
      if p_connection > r
        add to connected neurons list
```

Listing 6.4: Pseudocode implementation of 3D reservoir structure generation.

This calculation provides us with a list of connections, which are based on distance. In order to represent these distances in a computational model, we employ simulated axonal delays. In effect, by deliberately delaying the normally instantaneous spike transmission exhibited *in silico*, we represent the slower axonal transmission exhibited *in vivo*. These delays should be proportional to the axonal length (*i.e.*, the inter-neuron distance, d_{ij}). However, in the case of some neuromorphic hardware systems, constraints on such delays are imposed. In both the SpiNNaker and Zheijiang systems, the maximum axonal delay is 16 ms. Section 7.4.3.1 discusses how these axonal delays should be calculated in the case where they are constrained by the simulation system.

This method of network generation has a time complexity of $O(n^2)$ where n is the number of neurons, and is therefore not appropriate for large networks. It would be straightforward to perform this calculation using matrices if larger networks were required. However, for the network sizes it has currently been applied on, this implementation is sufficient, and has deliberately been implemented in as straightforward a way as possible, for readability and the understanding of the user. Some performance improvements are discussed in the next section.

6.4.3.2 Just-In-Time Compilation of Large Loops

As mentioned prior, this type of calculation has a rough complexity of $O(n^2)$ where n is the number of neurons. As a result, for large numbers of n the calculation can be slow.

```
import numba
# *subsequent lines omitted here*

@numba.jit # activate JIT compilation of the following method
def calculate_connection_matrix(self, inhibitory_split=0.2,
    connection_probability=0.025):
  for i, presynaptic_pos in enumerate(self.positions_list):
  # *subsequent lines omitted here*
```

Listing 6.5: Simple annotation of Numba JIT to standard Python.

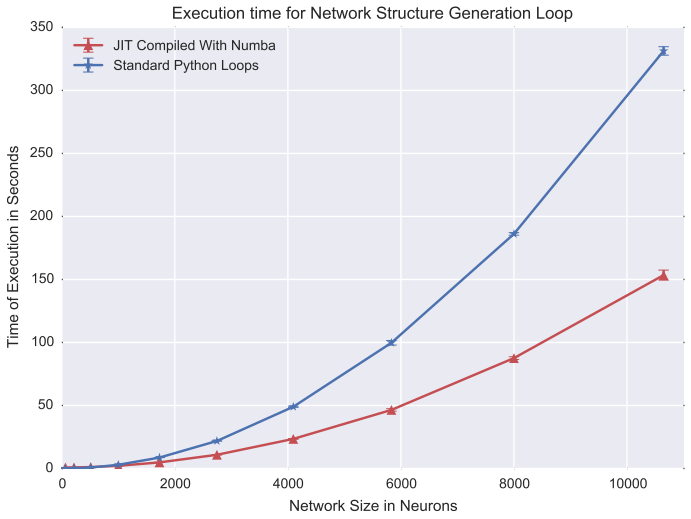
To identify the extent of this issue, profiling on a number of network scales was performed. Here, the `kernprof` library[4] has been used to time the execution of method calls in the `NeuCubeReservoir` class, which is responsible for the setup and operation of the large-scale structured reservoir. An example of such a report is too long to print here, but a representative copy is available online.[5]

Firstly, a series of arbitrarily shaped (cuboid) networks at a roughly linear scale with regard to the network 'sides' were generated. These networks were then run three times each in the PyNN version of the NeuCube, on the Brian simulation platform. The results of this experiment are shown as the blue (starred) traces in Figure 6.3. From the results of this experiment, it is clear that the performance of a pure-Python nested loop for the generation of a network structure is inefficient. This is in line with the general performance of Python, in that looping operations are generally slow and should be avoided where possible. However, as our intent here is to make the operation of the NeuCube as transparent as possible, it is undesirable to move this calculation to a complex set of vector operations – more efficient though that may be.

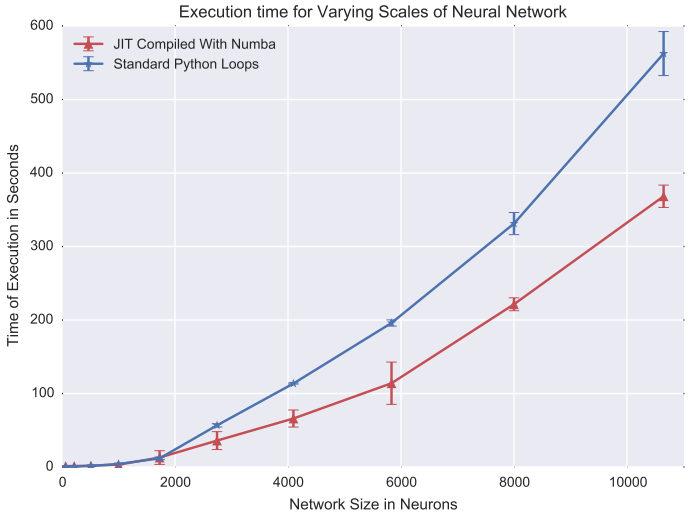
In order to address this issue, we look to performance improvements for native Python code. The `numba` library[6] uses the LLVM[7] compiler to just-in-time compile Python code to native machine operations, bypassing the normal Python interpreter. As a result, it is possible to significantly improve the execution speed of mathematical and looping operations.

The addition of Numba to our system is straightforward. In the simplest case, we firstly `import numba`, and annotate the declaration of the method we wish to optimise, as described in Listing 6.5.

[4] https://github.com/rkern/line_profiler
[5] https://gist.github.com/nmscott/ec48d0fe2fd6928d5f95
[6] http://numba.pydata.org/
[7] http://llvm.org/



(A) Execution time performance increase for the Network Structure generation loop.



(B) Execution time performance increase for whole NeuCube reservoir system.

FIGURE 6.3: Performance improvements found when applying Just-In-Time compilation to the Network Structure generation loop. Here we see a speedup in both relative and absolute terms, for the loop itself (Figure 6.3a) and the application as a whole (Figure 6.3b). In both cases, blue (starred) markers represent standard Python loops, red (caret) represents those JIT-compiled with Numba.

Our earlier profiling experiment was repeated for this alteration. The results are shown as the red (caret) trace in Figure 6.3. There is a significant performance increase found in JIT compilation of this single method. This method executes in around half of the time of the ordinary Python version, with only one line of code added, and still operating in native Python code. This performance increase is sufficent for these scales of networks, but as discussed in the above section, a matrix-based

implementation should be sought for much larger networks.

The MATLAB ('M1') implementation of the NeuCube has not been compared here, as it functions in a fundamentally different manner, and for a fundamentally different application context, to the PyNN/Python version. The inherent issues in the implementation of the M1 model alone would be sufficient justification to render a direct comparison void. Additionally, the PyNN implementations are complicated by their use of a neuromorphic hardware system to perform the simulations. A more meaningful comparison would be the difference in performance between the different simulation platforms utilised by the PyNN implementation; this is presented in further detail in Section 7.5.

6.4.3.3 Input Location Mapping

Input location selection for the reservoir is of critical importance in the NeuCube, as it is here that the most significant contribuition to representing the underlying spatial or spectral distribution of the data is made. However, it is also a difficult task to map the at-times inaccurate or general input locations into the specific space of the NeuCube reservoir. If we have a general input location l_i which may or may not have a corresponding neuron location $l_n \in \mathbf{L}$ where $\mathbf{L}$ is the list of neuron locations in the NeuCube reservoir, we need some method of searching for l_n (or, at least its closest match). We can use a data structure called a k-d tree for this task.

In straightforward terms, a k-d tree is an extension on a binary tree, where each node in the tree is a point in k-dimensional space. More formally, it provides a binary space partitioning tree-style decomposition for a point set, where the point set can be of any dimension.

Informed by its introduction in Bentley (1975), the generation of a k-d tree will be explained in general terms. Non-leaf nodes (*i.e.* those which have sub-nodes) can be considered to implicitly generate a splitting hyperplane that divides the k-dimensional space into two sub-spaces, named in the literature as 'half-spaces'. Points to the conceptual left of this hyperplane are represented by the left subtree of that node, and points conceptually right of the hyperplane are represented by the right subtree. To choose the hyperplane direction, we associate every node in the tree to one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. In its worst case, a k-d tree search has a time complexity of $O(n)$ and an average case $O(\log n)$. The worst case is seen only when $k \to n$, *i.e.* that the number of dimensions

approaches or is greater than the number of elements to search. As we are only searching in a maximum of three dimensions, we should never reach this worst case.

As we have a relatively straightforward use case (searching a constrained 3D space for a single point or neighbourhood of points), the optimised function provided in `scipy.spatial.KDTree` has been used, which follows the work of Maneewongvatana and Mount (2001).

Interestingly, as traversing the k-d tree can be thought of as a nearest-neighbour search with a neighbourhood of one (*i.e.* a kNN where $k = 1$), it is trivial to extend this algorithm to provide an arbitrary neighbourhood size. Instead of searching for the single closest neuron i in the k-d tree tree to our search location, we can instead simply select the nearest n locations and discover the neighbourhood of that search location. This feature has been implemented in the system introduced in this chapter, although it is not yet in use. This feature may be useful in the future, as it will allow us to select a neighbourhood of neurons to represent a single input, in much the same way that an EEG represents an area of cortical activity. This may have an impact on the performance of the unsupervised learning in the NeuCube reservoir, as it will increase the number of neurons that are activated with a single input channel. This may be an effective way to 'boost' a weak signal. Exploration of this behaviour, particularly in the context of spatially-disperse signals such as seismograph data, is warranted.

6.4.4 Inconsistencies Between PyNN and MATLAB Versions

A basic implementation of the NeuCube was intially developed in MATLAB by two students visiting KEDRI. This version, with a number of additions and improvements, has subsequently become the *de facto* prototying and testing system for general users of the NeuCube framework. It is identified as Module 1 in Figure 6.1, and will hereafter be referred to as M1.

Unfortunately, due to the expedited manner in which this system was written, the software exhibits a number of inconsistencies with the supporting theory. The architecture of M1 here will not be discussed here in any detail, except to say that it is monolithic and an effort is now being made to refactor this code with modularity and extensibility in mind, following the principles established in this chapter.

Three primary issues in M1 were identified in the development of this thesis:

1. An implementation of the unsupervised learning in the NeuCube reservoir which diverges from the literature (Section 6.4.4.1);
2. An alternative implementation of excitatory and inhibitory synapses and neurons (Section 6.4.4.2); and
3. A lack of 'distance' (axonal delays) in the model (Section 6.4.4.3).

These issues and how they impact the subsequent development of NeuCube models and behaviours will be introduced here. Inter-module communication (Section 6.3.3) is also directly affected. Due to the inconsistencies discussed herein, a model trained in M1 cannot at present be directly transferred to a different implementation of the NeuCube.

This is not to say that the existing system does not work – simply that it works in a slightly divergent manner to the way it was intended and described in the theory. This MATLAB version does produce results in accordance with most of the basic principles of the NeuCube framework. It is, however, a simplified form of most of these principles. The behaviours of the MATLAB version will likely be different to any alternative versions (*e.g.* the PyNN implementation in Section 6.4) for the reasons introduced in this section. The NeuCube Core architecture in Section 6.6 introduces one way we seek to address this issue in the non-PyNN implementations of the NeuCube framework.

6.4.4.1 STDP Implementation of MATLAB Version

Unsupervised learning is a key component of the NeuCube framework. The unsupervised learning implementated in the MATLAB M1 version of the NeuCube can be described as STDP 'inspired', rather than a strict implementation of the learning rule as it is described in the literature. The Python/PyNN version of the NeuCube introduced in this thesis uses the canonical form of STDP introduced in Section 3.6.1.1 and discussed in the context of the SpiNNaker device in Section 7.4.2.3.

In effect, the learning rule as implemented in the M1 approximates a spike-rate proportional weight increase, based on the spike rate of the postsynaptic neuron only. A pseudocode implementation of this alternative learning rule is given in Listing 6.6. Here we will refer to this as Rate Dependent Synaptic Plasticity (RDSP) in this section.

For clarity, the updated synaptic weight w' calculation is expanded here rather than in the pseudocode,

$$w' = w_{ij}(t_{\text{last}}) - \frac{\alpha}{t - t_{\text{last}} + 1} \tag{6.2}$$

where w is the current synaptic weight; t is the current time point; t_{last} is the last time the postsynaptic neuron spiked; and α is a scaling function.

Of interest to us is the fact that this weight update formula is divergent from that of STDP (*cf.* Section 3.6.1.1). That is not to say that it is incorrect: merely that it is different from the implementation suggested in the literature.

```
for (presynaptic neuron i = 1 to size(input))
  for (postsynaptic neuron j = 1 to size(output))
    if connection_ij
      w_ij(t) = w'
```

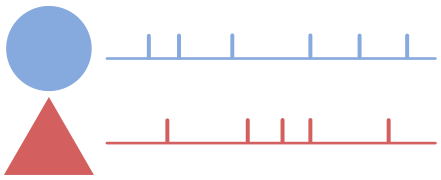
Listing 6.6: Pseudocode implementation of STDP-style Unsupervised Learning in M1 Reservoir.

In the case where a model is trained in M1 using this RDSP, an otherwise identical model trained in the PyNN implementation will converge to drastically different synaptic weights. Different synaptic matrices will of course lead to different network dynamics, and would render a model originally trained in the M1 unable to train further in alternative implementations of the NeuCube. A mapping of the connectomes from one module to the other would still be possible, but further cross-platform training of the reservoir would be meaningless.

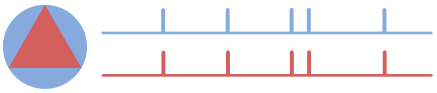
Functionally, the results of the training process are similar between the RDSP introduced here and the canonical STDP, with the primary difference in the magnitude of the weight changes. RDSP will not limit the maximum synaptic weight over the lifetime of the network, and as a result, a network trained with this algorithm will most likely have higher levels of spiking activity at the end of a simulation than an identical network trained with STDP.

6.4.4.2 Excitatory and Inhibitory Synapses and Neuron Populations of MATLAB Version

Recall that in biological neural networks, there are two basic types of neuronal dynamics; excitatory and inhibitory. In basic terms, excitatory and inhibitory neurons increase or decrease the membrane potential of their postsynaptic neurons, respectively. These neurons act to maintain a balance of activity within the network, preventing either too many spikes (leading to neuronal saturation), or too few spikes



(A) Separate excitatory (top, blue, circular) and inhibitory (bottom, red, triangular) populations.



(B) Combined excitatory (top, blue, circular) and inhibitory (bottom, red, triangular) populations.

Figure 6.4: An illustration of the synapse and neuron population dynamics implemented correctly (Figure 6.4a) in the PyNN version, and incorrectly implemented (Figure 6.4b) in the M1 version. Note that in Figure 6.4a the neuron spiking behaviours between excitatory and inhibitory are uncoupled and may be significantly different, whereas in Figure 6.4b they are identical.

(leading to a silent network). In biological terms, this is the difference between an epileptic state or death. This homeostasis is of vital importance in models of SNN, and is therefore generally included. Typically, we would find that the excitatory/inhibitory division of neurons in a large network is on the order of 80% excitatory and 20% inhibitory. In computational models, these are often distributed randomly in the network, unless following some *a-priori* biological pattern such as that seen in the human visual cortex. See Section 3.2 for further details of the biological basis of this property.

Currently, M1 represents excitatory and inhibitory neurons as one homogeneous unit. That is to say, a single neuron is both excitatory *and* inhibitory. This is achieved through the simple expedient of positively- and negatively-weighted synapses representing excitatory and inhibitory connections, respectively. Implementing excitation and inhibition in this manner is not unheard of. For example, see the Zheijiang University FPGA described in Section 7.2. However, it is certainly uncommon in, and unsupported by, the established body of theory, with respect to biological plausibility. See Figure 6.4 for a visual intuition of the principle described here. Figure 6.4b shows the combined neurons identified in the M1, and Figure 6.4a shows the desired separate populations as used in the PyNN version (*cf.* Section 6.4).

Such a simplification leads to an undesirable situation, where a single presynaptic neuron excites its postsynaptic neurons connected via excitatory synapses and simulataneously inhibits those connected via inhibitory synapses, with identical spike trains. This means that as a neuron is more excitatory, it is also more inhibitory, in exactly the same proportions. In turn, this property affects the network's ability

to self-regulate, as proportionally equivalent excitation and inhibition may not be desired in the network at that time. Just as significantly, this behaviour affects the unsupervised training present in the network as it is then learning the wrong spike trains.

In an ideal case these excitatory and inhibitory neurons would be entirely separate, *i.e.* a population of excitatory neurons with only excitatory synapses, and a population of inhibitory neurons with only inhibitory synapses. Each of these populations would be connected both within itself, and to neurons in the other population. Generally speaking, these populations would also have different properties in terms of neuronal and synaptic dynamics.

This ideal case is in fact how the vast majority of SNN simulation systems, including networks defined with PyNN, require their networks to be defined. The systems introduced in this chapter, particularly that of the PyNN implementation in Section 6.4, all use this system of neuronal and synaptic dynamics.

When looking at model portability between NeuCube implementations, this inconsistency is an obvious issue. It is impossible to map a trained model developed in M1 to any other implementation, including the PyNN version introduced prior. Two possible mitigation strategies for this issue and their inadequacies are discussed below.

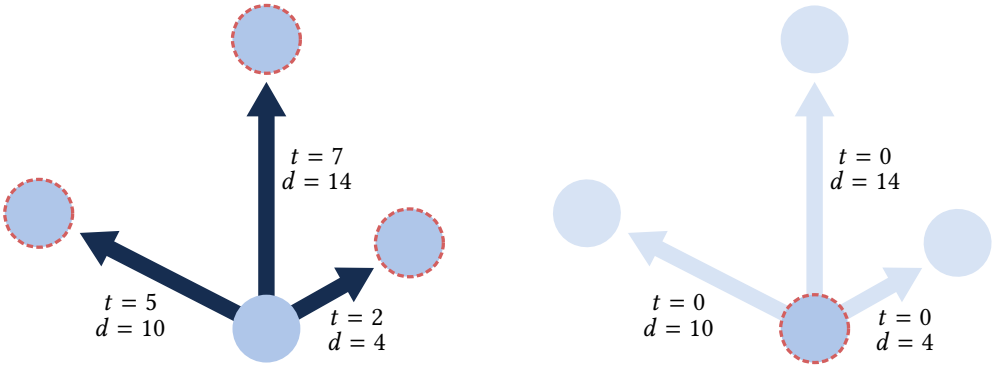
1. Simply map the neurons and their connectome directly;

This is not possible, as there is no ability for a neuron to be simulataneously excitatory and inhibitory in the vast majority of simulation systems, including those supported by PyNN.

2. Create two neurons (one excitatory and one inhibitory) in the same physical location to represent a single M1 neuron;

These new neurons will then have separated the pre- and post-synaptic populations and therefore have different membrane potentials at different times, leading to the trained synaptic weights being meaningless and in either case, different spiking behaviour from the two populations.

Until this oversight is corrected it will not be possible to directly port a model from the M1 module to the PyNN module, and vice-versa. Section 6.4.3.1 discusses how these populations should be generated in the future.



(A) 3D network structure with distance proportional axonal delays (all other NeuCube versions)

(B) 3D network structure without distance proportional axonal delays (M1 version)

Figure 6.5: An illustration of the inconsistency in axonal delay implementations between the PyNN implementation (Figure 6.5a) and M1 (Figure 6.5b). Blue (solid) circles represent the physical location of neurons in the network. Red (dashed) circles represent their positions as they are conceptually located in the simulation. Axonal delay t should be proportional to the physical distance d of that connection. In M1, $t \forall = 0$, whereas in reality (as in the PyNN version) $t \propto d$. Values of t and d are arbitrary and for illustration only.

6.4.4.3 3D Structure and Conceptual Distances of MATLAB Version

A vital aspect of the NeuCube framework is the concept of a 3D, meaningfully structured network. Without this structuring, the NeuCube is effectively a traditional LSM with STDP. Typically, in the simulation of spiking neurons, the distances between the neurons (and therefore, their structure) is represented as a propagation delay between two neurons. This delay then represents the time a signal should take to propagate along the length of the axon between pre- and post-synaptic neurons.

Axonal delays are handled differently depending on which simulation platform is used by your model. In almost all cases, the physical propagation of the signal from one conceptual neuron to another *in silico* is negligible – in most cases, less than microseconds. Actual axonal delays then, must be simulated in the simulation system in some manner. Typically these delays will be represented by a queue of incoming spike times (PyNEST, Brian) or rotating buffer (SpiNNaker, *cf.* Section 7.4.3.1) that takes the incoming spike time and adds a delay proportional to the axonal distance. After waiting until this time (to simulate transmission time) the spike is propagated to the postsynaptic neuron. In this way, a complex 3D structure can be conceptually represented by a simple series of matrices.

No delays are implemented in networks simulated in M1. This means that conceptually the neurons have no distance between them, and therefore, no 3D structure. The 3D locations of the neurons are used initially in the generation of the connectome, but in the actual simulation of the network dynamics spike transmission is functionally instantanous, *i.e.* that the postsynaptic neurons universally receive the spikes by the next timestep regardless of their actual distance from the presynaptic neuron. See Figure 6.5 for a visual intuition of this issue.

Consequently, the STDP learning in M1 will learn the spike trains for a network of neurons with effectively instantaneous (and universally equal) spike transmission, and not those that would represent a network with distance dependent axonal delays.

In the NeuCube versions introduced in this thesis, these axonal delays exist and are proportional to the distance between neurons. Axonal delays are necessary, as mentioned above, to functionally incorporate the concept of distance and 3D structure in these networks. Without this 3D structure, the NeuCube lacks the ability to conceptually map SSTD in a meaningful way.

Therefore, we cannot directly transfer a model trained in M1 to any of the other NeuCube versions, due to the fact that the trained synaptic weights in such a model are trained for *instantaneous* transmission delays, and not *distance dependent* transmission delays. The correct method of calculating axonal delays is given in Section 7.4.3.1, and was also briefly discussed earlier in this chapter, in Section 6.4.3.1.

6.5 Position of NeuCube M1 Module and its Future

Given that a number of issues in the M1 (MATLAB) version of the NeuCube have been made evident in the previous sections, its position in the development of the NeuCube – and particularly, its position in this thesis – should be clarified.

The M1 version of the NeuCube is presently the dominant implementation of the NeuCube as it exists in the literature. This justifies its discussion in this thesis, in the contexts it has been introduced. We can consider the M1 NeuCube to be a prototype implementation, with some issues to be rectified.

It is also difficult to make a great deal of direct comparisons between this implementation and the PyNN implementation introduced in this thesis. Due to the issues discussed in the previous sections (*cf.* Section 6.4.4), and the fundamentally different computational platforms, we cannot directly compare performance or network behaviours across implementations other than their direct outputs.

We can reasonably state that while the MATLAB implementation of this system has a number of identified incosistencies, it is not, as a whole, incorrect. Its effectiveness in the case studies presented in the literature attest to this.

While the M1 NeuCube is discussed in this thesis and comparisons are made in some contexts, this implementation is used as a reference only where it is pragmatic and meaningful. It is not intended that the M1 should be considered a general reference implementation; in fact, one of the intents of this work is to introduce a more general purpose and consistent implementation into the literature.

6.6 NeuCube 'Core' Architecture

We have introduced a number of issues in terms of cross-platform compatability and implementation differences leading to discontinuities in the dynamics of a NeuCube model in the previous sections. These challenges, along with the realisation that we are duplicating efforts and 'reinventing the wheel', has led to the conception of the NeuCube Core system. This system is still in its infancy, and as a result, only an overview can be given here.

The Core is a novel architecture for the efficient implementation of NeuCube systems across a number of platforms. Developed by the author of this thesis, S. Marks, and N. Sengupta, the Core is still very much in the planning stages; as yet, no code has been written. However, it has been developed from first principles to better address the issues of scaling, extensibility, and multi-platform multi-language support which the NeuCube will require in future applications.

The basic concept of this approach is the development of a Core[8] system to provide the functionality for simulating neuronal and synaptic dynamics in an efficient way. See Figure 6.6 for a schematic overview of this architecture. This Core will likely be developed in C++. Access via our typical high-level modelling languages (Python, MATLAB, *etc.*) will be provided through language-specific wrappers on the Core API. Facility for add-on modules such as the CNGM and personalised modelling systems identified in the greater NeuCube architecture (Figure 6.1) to interface with the model dynamics calculations in the Core will be provided. A similar facility will be provided for the immersive visualisation system discussed in Section 4.5.2. At a lower level, a hardware interface layer will provide support for cross-platform deployment of NeuCube systems on standard computing hardware (including AWS/Cloud systems

[8]The terms Core and 'Core' are distinct concepts; the Core is the overarching architecture, while the 'Core' is the specific module of that architecture responsible for network dynamics calculation

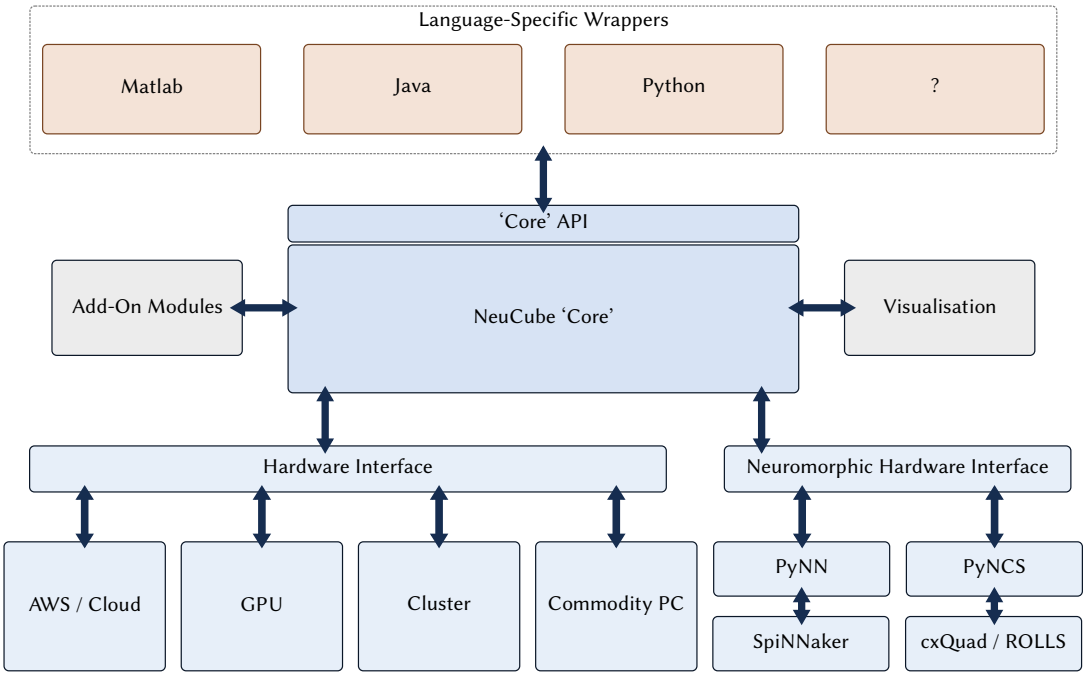


Figure 6.6: Block diagram of the proposed NeuCube Core software architecture. The central structure is the NeuCube Core, which will provide a thoroughly tested and optimised development of neuronal and network dynamics for the simulation of NeuCube architectures, which functionality will be provided through an API. The topmost boxes are language-specific wrappers for the methods the NeuCube Core API provides. To the left are the add-on modules such as personalised modelling. To the right is a direct interface to the immersive visualisation tool. To the bottom we see the hardware interface layers; these provide control and optimisation facilities for the Core system on a variety of different hardware systems.

and clusters). A *neuromorphic* hardware interface layer will pass-through commands from the API to neuromorphic systems like the SpiNNaker or analog VLSI. It can therefore subsume the SpiNNaker implementation of the NeuCube introduced in Section 7.4.

It is our intention that this system will eventually supplant the current development practices. As evidenced by the issues in Section 6.4.4, there is considerable potential for mistakes when developing such systems in a siloed manner. There is no sense in implementing multiple versions of the same neuronal and synaptic dynamics; this is inefficient, and draws developer attention away from implementing the specific features they are researching. If the basic features are provided for by the Core, development in this context can move away from this duplicated effort and towards meaningful new features. This type of system ensures that the network dynamics are traceable and predictable. We can ensure that a specific extension of the NeuCube utilises identical neuronal and synaptic model dynamics. Additionally, in this

way, if some advances are made to the efficiency or features of the Core, they are immediately available to all subsystems. Bug fixes and feature development are eased through this architecture, particularly when source and version control such as that identified in Appendix D is applied.

With the theoretical advantages it provides, we identify the expansion and implementation of this system as a future work of this thesis.

6.7 Chapter Summary and Conclusion

In this chapter, a number of major contributions to this thesis have been introduced. Firstly, an abstract design philosophy for the development of NeuCube systems, based on the Unix philosophy, is discussed. This philosophy emphasises aspects of modularity and simplicity in the software we create. With this in mind, we have then introduced a general software architecture and implementation framework for the NeuCube systems. This draws on aspects of object-oriented design and the Template Method, and provides a foundation for the later introduction of a concrete implementation of this framework. Considerations for integration with existing systems were introduced, including the use of AER for streaming data. A realisation of the previously introduced framework using Python and PyNN was introduced. PyNN has been chosen here due to its straightforward model definition, cross-platform simulator support, and additionally because it will support certain neuromorphic systems. This system was comprehensively discussed, including challenges in its development. Finally, an alternative software architecture for the integrated development of NeuCube systems known as the Core was introduced, which is intended to further centralise and speed development of NeuCube systems.

Contributions of this Chapter

1. The introduction of a specific 'design philosophy' for implementations of the NeuCube architecture.
2. A general architecture for the implementation of NeuCube software systems in an object-oriented context.
3. A reference implementation of the NeuCube framework developed in PyNN.
4. The identification of a theoretical improvement in the adoption of the Strategy design pattern, which should facilitate automated optimisation of the NeuCube.
5. Identification of a research area in the use of a neighbourhood of neurons to represent one input channel.

6. Identification of issues in the MATLAB M1 implementation of the NeuCube and how this affects the integration with the new PyNN implementation.
7. The introduction of the Core architecture for integrated development of NeuCube systems.

Peer Reviewed Literature With Direct Contributions from this Chapter

1. Kasabov, N., **Scott, N. M.**, Tu, E., Marks, S., Sengupta, N., Capecci, E., Othman, M., Doborjeh, M., Murli, N., Hartono, R., Espinosa-Ramos, J.I., Zhou, L., Alvi, F., Wang, G., Taylor, D., Feigin, V., Gulyaev, S., Mahmoud, M., Hou, Z.-G. and Yang, J. (2016). Evolving Spatio-Temporal Data Machines Based on the NeuCube Neuromorphic Framework: Design Methodology and Selected Applications. *Neural Networks*. Special Issue on Learning in Big Data. Elsevier. doi:10.1016/j.neunet.2015.09.011
2. **Scott, N. M.**, Mahmoud, M., Hartono, R., Gulyaev, S., and Kasabov, N. (2015). Feasibility analysis of using the NeuCube Spiking Neural Network Architecture for Dispersed Transients and Pulsar Detection. In *13th International Conference on Neuro-Computing and Evolving Intelligence*. February 19–20, Auckland, New Zealand.
3. **Scott, N. M.**, and Kasabov, N. (2015). Feasibility of Implementing NeuCube on the SpiNNaker Neuromorphic Hardware Device. In *Proceedings of the 13th International Conference on Neuro-Computing and Evolving Intelligence*. February 19–20, Auckland, New Zealand.
4. **Scott, N. M.**, Kasabov, N., and Indiveri, G. (2013). NeuCube Neuromorphic Framework for Spatio-Temporal Brain Data and Its Python Implementation. In *Proceedings of the 20th International Conference on Neural Information Processing*, 3–7 November 2013, Daegu, Korea. Springer. doi:10.1007/978-3-642-42051-1_11

Chapter 7

Neuromorphic Hardware Implementations

> The brain is a monstrous, beautiful mess. Its billions of nerve cells lie in a tangled web that displays cognitive powers far exceeding any of the silicon machines we have built to mimic it.
>
> — William F. Altman
> (*Apprentices of Wonder*, 1989)

Neuromorphic hardware as a field of study attempts to model the complex dynamics of biological neurons in some physical circuit, be it digital or analog. Such hardware-based implementations of SNN are currently undergoing something of a renaissance, due to a number of factors. Primary among these is the consideration that the human brain has a remarkable power efficiency – somewhere on the order of 20 Watts – for its ability to learn and operate in an uncertain environment.

Certain applications which have for one reason or another become hugely important in recent times, including neural prosthetics and autonomous robotic control, are particularly appropriate for SNN. These opportunities are difficult to take advantage of when we are limited to software emulation of SNN behaviour. Significant concerns here include power consumption and heat production of such circuits, which are serious impediments to robotic implementations or implanting neural prosthetics in humans. Additionally, neuromorphic systems generally operate in real-world time, which is of interest to us when SNN systems interact with real world stimuli. Here, it is meant that there is a 1:1 mapping of simulation (device) time to wall-clock time; *i.e.* that if we simulate a network for 1 s of device time, this will take 1 s of wall-clock time to complete.

The motivation for this thesis' interest in such an area is the theoretical applicability of a framework like the NeuCube in the domains which neuromorphic systems excel; primarily, where noise-tolerant, adaptive, low power, fast learning is required. In addition, it is impossible for the developers of such hardware systems to demonstrate the practical viability of their respective approaches without an effective and complementary learning system. Here, we introduce these main families and in a later section describe considerations for implementing the NeuCube on the same.

7.1 A Review of Neuromorphic Hardware Systems

Motivating neuromorphic hardware development are a number of factors. Current literature suggests that ongoing enhancement of traditional von Neumann architectures are not likely to reduce simulation runtime significantly, as single and multi-core scaling face their limits in terms of transistor size (Thompson & Parthasarathy, 2006), energy consumption (Esmaeilzadeh, Blem, St. Amant, Sankaralingam & Burger, 2011), and communication (Perrin, 2011). Neuromorphic hardware systems are an alternative to such systems that have the potential to alleviate these limitations. Their underlying circuits are especially designed to solve neuron dynamics and can be highly accelerated compared to biological time (Belatreche, Maguire & McGinnity, 2006). The interested reader is directed to the reviews in Indiveri et al. (2011) and Furber (2016) for further details.

Hardware implementations of neuromorphic systems began at CalTech during the late 1980's, in the laboratory of Carver Mead. He was the first to recognise the ability of transistor-capacitor circuits to accurately simulate the spiking behaviour of biological neurons. See Mead (1989) for an example of his early work. Since then, a vast number of different neuromorphic systems have been trialled. Such systems can be broadly categorised in one of the families of Application-Specific Integrated Circuit (ASIC), Field Programmable Gate Array (FPGA), or digital systems. Here, we will introduce these main families and in later sections describe considerations for implementing the NeuCube on the same.

A visual intuition of these devices (including software simulation on commodity hardware) with regards to the trade-offs between power consumption and performance is shown in Figure 7.1, and between power consumption and model flexibility in Figure 7.2. A tabular comparison has been given in Table 7.1. In general, as the hardware systems become more specific and dedicated (*i.e.*, towards the direction

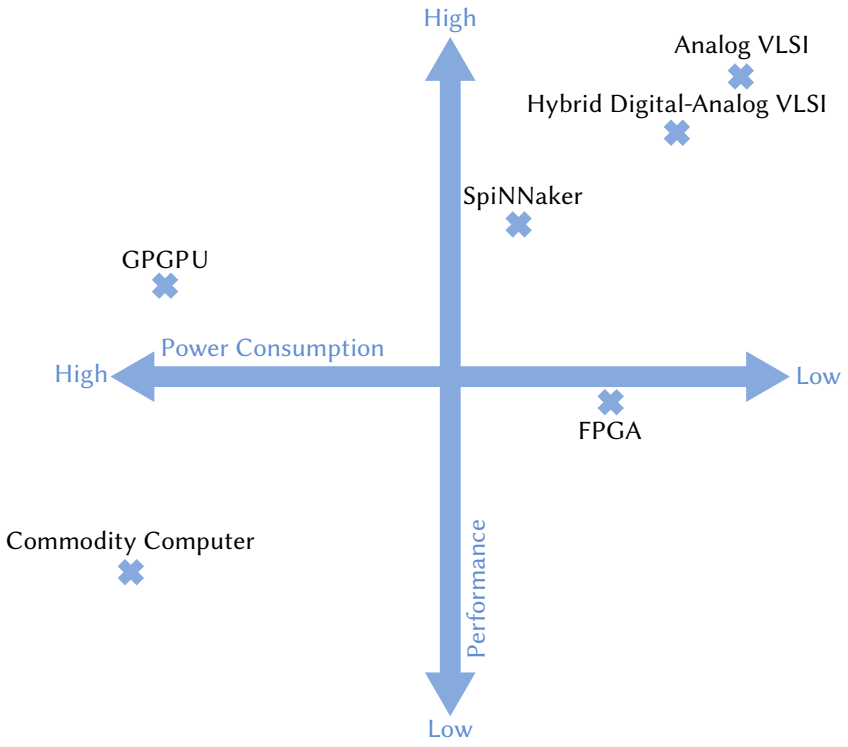


Figure 7.1: Intuition of tradeoff between power consumption and relative performance in neuromorphic systems. Depending on the underlying goal of the specific hardware, relative computational cost for the same network will *generally* be lower on lower powered systems, as these have dedicated circuits for the calculation of neuronal and synaptic dynamics and less process overhead. This of course depends on the specific hardware device and network chosen. The baseline is shifted to emphasise that even commodity computing solutions are *relatively* effective at simulation of such networks.

of ASICs), both power consumption and computational cost is reduced, with a proportional reduction in the model and network selection choices. For example, a subthreshold analog VLSI ASIC will have remarkably low power consumption, at the cost of not being able to change the neuronal model. We note that this is not a negative characteristic; merely that it is a consideration when making a selection from the available hardware.

When discussing neuromorphic hardware, it is interesting to mention an unique constraint. Manufacturing tolerances of analog hardware can account for up to ±20% variability in network parameters such as the firing threshold, leak rate, and synaptic weights, which are typically fixed (G. Indiveri, personal communication, April 2012). This is also the case for the FACETS system (Pfeil et al., 2012) and most VLSI. The effect of this 'device mismatch' is particularly evident as circuit density increases

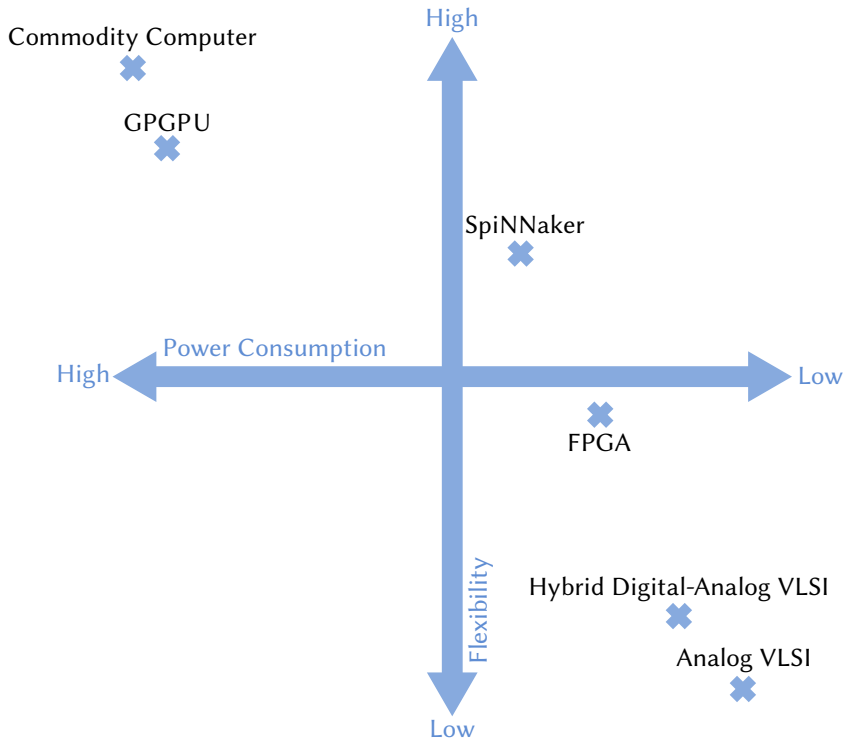


Figure 7.2: Intuition of tradeoff between power consumption and relative model flexibility in neuromorphic systems. Typically, lowered power consumption is correlated with lower flexibility in terms of model design and selection, as there is a higher liklihood that such models are specifically implemented in the hardware.

and the relative sizes of the components shrink. However, with normal learning algorithms, the accuracy of learning is not affected by the asymmetry between synapses that may come as a result of this hardware variablity as it can be 'learned away'. Bear in mind that software simulations typically start with some randomness in at least their synaptic weights, if not other parameters (*cf.* Section 3.7.2 and 4.2).

In order to implement these systems with the lowest die (circuit) area and lowest power consumption, some further constraints are imposed, particularly with respect to synaptic weights. The synapic weights of FACETS hardware have a 4-bit resolution (Pfeil et al., 2012). Other networks employ bistable 1-bit resolution synapses (Badoni, Giulioni, Dante & Del Giudice, 2006; Indiveri et al., 2010). Bistable synapses are sufficent for memory formation (Amit & Fusi, 1994; Brader et al., 2007; Fusi, Drew & Abbott, 2005). However, in some cases these models do not only use the spike timings to calculate the weights (Bi & Poo, 2001; Markram, Lübke, Frotscher & Sakmann, 1997), but also the postsynaptic potential requiring additional hardware (Sjöström, Turrigiano & Nelson, 2001). Bistable synapses are supported in computational models

Table 7.1: Comparison table for key features of the major neuromorphic systems explored in this thesis. Details for the SpiNNaker and TrueNorth devices are from the table in Furber (2016), which provides an excellent review of modern neuromorphic systems.

	Platform			
Feature	SpiNNaker	Zheijang FPGA	TrueNorth	cxQuad
Type	Programmable Digital	FPGA	Fixed Digital	Subthreshold ASIC
Neuron model	Programmable	LIF	LIF	AIAF
Synapse model	Programmable	Programmable	Binary	Mixed
Max # Neurons				
– (Per Element)	16K	2048	1M	1024
– (Planned System)	460M	Not Defined	Arbitrary	Not Defined
Max # Synapses				
– (Per Element)	14M	4.2M	256M	64K
– (Planned System)	460B	Not Defined	Arbitrary	Not Defined
Runtime Plasticity	Programmable	No	No	Programmable Biases
Energy Per Connection	10 nJ	Unknown	25 pJ	Unknown
Routing Model	Mesh Multicast	Mesh Multicast	Mesh Multicast	Mesh Multicast
Biological Speedup	1x	1x	1x	1x

as well as in hardware. If the probability of a transition between states is low, and long term potentiation is balanced against long term depression, optimal storage capacity in terms of its behaviour as a palimsest is retained, even with two state (bistable) synapses (Amit & Fusi, 1994; Fusi, 2000). Therefore, this property of the hardware is not a limitation in theoretical terms. In practical terms, we must be careful when developing learning algorithms and networks for these hardware systems that they function correctly with the relevant synapse/neuronal parameter limitations. A further work identified in this thesis and discussed in Section 7.3 regards this in the context of the NeuCube.

7.1.1 Field-Programmable Gate Array Neuromorphic Systems

A Field Programmable Gate Array (FPGA) is an integrated circuit designed to be configured by end users after its initial manufacture. The FPGA configuration is generally specified using a Hardware Description Language (HDL), and can be used to implement any logical function that an ASIC could perform. The ability to update the functionality after shipping, configuration of design aspects and the low non-recurring engineering costs relative to an ASIC design, offer advantages for many applications. FPGAs contain programmable logic components known as 'logic blocks', and a hierarchy of reconfigurable interconnects that allow these blocks to be connected. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the

logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

Rapid design time, low cost, flexibility, digital precision, and stability are beneficial characteristics of FPGAs, which are encouraging their exploration as a promising alternative for designing neuromorphic systems. The general FPGA approach uses digital computation to emulate individual neuron behaviour, and a parallel and distributed network architecture with large numbers of chips to implement the system behaviour. In addition, it is possible to utilise off the shelf FPGAs. While digital computation consumes more silicon area and power per function than its analog counterpart, it is high precision, and has greater stability, reliability, and repeatability, as it is not susceptible to power supply and thermal noise, or device mismatch (Cassidy, Denham, Kanold & Andreou, 2007). Despite a lower level of biophysical realism, implementations of non-trivial neuro-biological functions such as speech processing (Cassidy et al., 2007), autonomous robotics control (de Garis, Korkin & Fehr, 2001), and detailed models of neural behaviour (Graas, Brown & Lee, 2004; Pearson et al., 2005) have been demonstrated in FPGAs. FPGA are also used for monitoring and board IO processes in the SpiNNaker hardware system described in Section 7.4.1. An implementation of the NeuCube on FPGA is discussed in Section 7.2.

7.1.2 Application-Specific Integrated Circuit and Very Large Scale Integration Neuromorphic Systems

Very Large Scale Integration (VLSI) is the process of creating integrated circuits by combining thousands of transistors into a single chip. Application-Specific Integrated Circuit (ASIC) devices are customised integrated circuits, designed for a particular task. Most VLSI ASIC devices (almost everything thought of as a 'microchip' or 'microprocessor') operate above threshold and are considered digital devices. By this, they operate at voltages above the gate voltage of the transistors present in the chip, leading to high precision in the representation of gating and binary data. However, transistors are also capable of operating in a sub-threshold capacity, and can be combined in a large scale, the result of which is known as analog VLSI. Remarkably, certain implementations of these analog designs can emulate the behaviour of biological neurons with a very high level of precision. These VLSI circuits operate using the same physics of computation used by the nervous system; *i.e.* they are silicon neuron circuits that exploit the physics of the silicon medium to directly reproduce the bio-physics of nervous cells (Indiveri et al., 2011), including complex behaviours

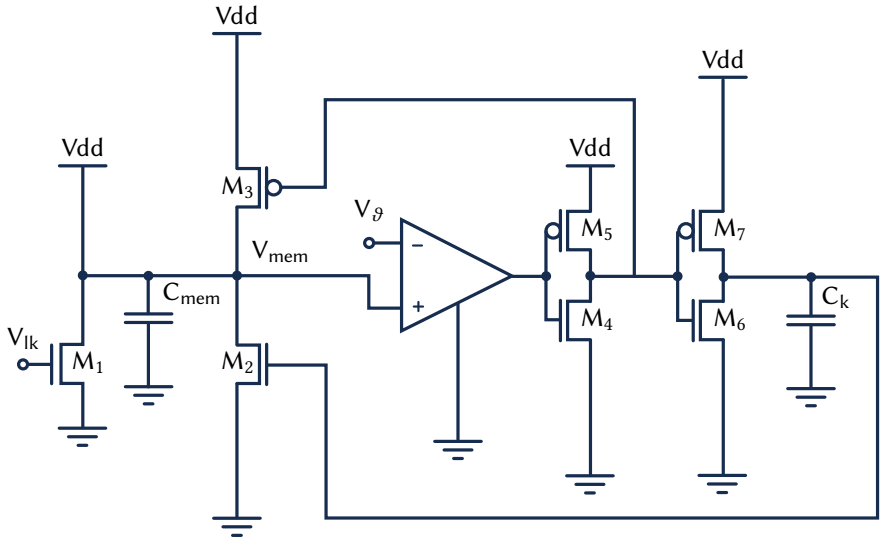
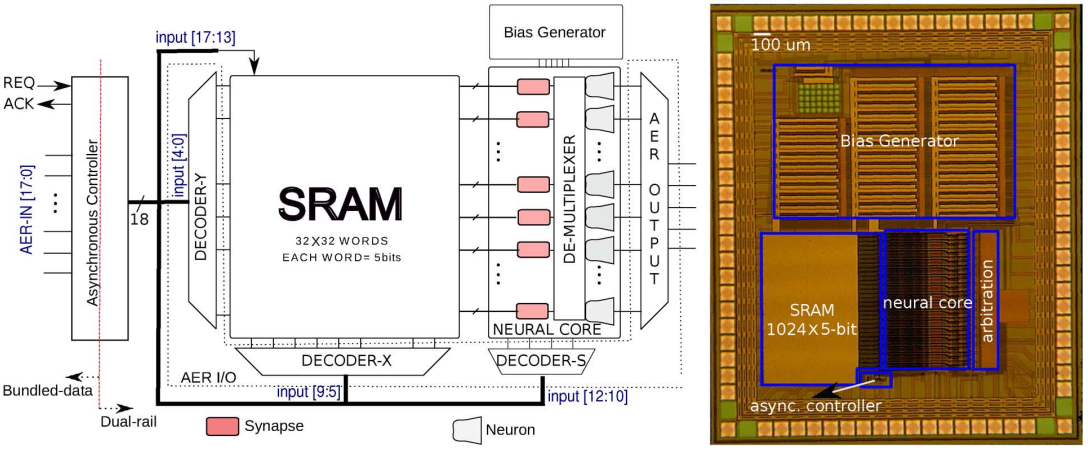


Figure 7.3: A circuit diagram of a simple spiking neuron realised in ASIC-style analog hardware recreated from Indiveri et al. (2011). This particular circuit recreates the dynamics of the Hodgkin-Huxley neural model. Such circuits are low power and occupy a small die area, so can be integrated on a massive scale at relatively low cost.

such as adaptive firing thresholds and Hebbian learning. An example of a single spiking neuron realised in hardware is shown in Figure 7.3, and an example of a whole spiking neuron chip (32 AdEx neurons) is shown in Figure 7.4.

VLSI is an important tool for investigating and implementing neural algorithms. It enables rapid prototyping of neural algorithms for the implementation of theories on neural computation, structure, network architecture, learning, and plasticity. Due to its high computational power, neuromorphic VLSI enables implementation of biologically inspired algorithms in real-time operation. This is of particular interest for sensory processing systems and biologically-inspired robotics. In the case of the NeuCube, this is of particular interest when we consider such end-use applications as BCI and neurorehabilitation.

Analog VLSI is advantageous in areas such as biophysical realism, density of neurons per chip, and low power, by comparison to digital circuits such as FPGA. It was shown in Kuon and Rose (2006) that designs implemented on FPGA need on average 40 times as much die (circuit) area, draw 12 times as much power, and are three times slower than the corresponding ASIC implementations. Analog VLSI systems have been used for simulation of biologically accurate Hodgkin-Huxley models (Yu & Cauwenberghs, 2010), neuromorphic vision sensors (Azadmehr, Abrahamsen & Häfliger, 2005; Lichtsteiner, Posch & Delbruck, 2008; Olsson & Häfliger, 2008), and learning



(A) Block diagram. (B) Micrograph.

Figure 7.4: The hybrid analog-digital SNN chip introduced in Moradi and Indiveri (2013). This chip comprises 32 AdEx neurons, each with 32x4 programmable synapses.

(Indiveri et al., 2010) among others. This wide range of applications, including the power and size advantages previously mentioned, indicate that analog neuromorphic VLSI is a promising area of research. The device discussed in Section 7.3 is one such area of research.

7.1.3 TrueNorth

Introduced in Merolla et al. (2014) – where it was the cover of that issue of *Science* – the TrueNorth chip of IBM is a result of their engagement in the DARPA SyNAPSE project. It comprises a wholly digital chip with 4096 neurosynaptic cores interconnected via an custom communication fabric. This architecture provides roughly 1 million programmable spiking neurons and 256 million configurable synapses per wafer. Wafers can be tiled in two dimensions via a communication interface, providing the capacity to scale the architecture to what they state is an arbitrary size, but is intended to be somewhere on the order of 10 billion neurons with 100 trillion synapses total (Merolla et al., 2014).

While a considerable amount of funding (≈ US$50 million) has been awarded to this project, it has yet to show empirical evidence of viability in realistic applications. Experiments applying the TrueNorth architecture to a number of different application areas (*e.g.* computer vision, sound recognition) were presented in Esser et al. (2013). However, these experiments were performed using their 'Compass' hardware simulator running on a standard BlueGene-P commodity supercomputer, and not on

the actual TrueNorth hardware itself. The use of hardware emulators for simulations of this complexity leads to the possibility of markedly different results between those presented in Esser et al. and identical experiments run on real hardware. While device mismatch (manufacturing tolerances) are not a significant source of variability in above-threshold digital systems, no software emulation of a hardware system can be 100% accurate. However, with this said, the TrueNorth architecture combined with IBM's financial and research resources suggests that it will likely become a viable system in the near future.

7.1.4 Memristor-Based Systems

Memristors, or memory-resistors, are a relatively recent development in neuromorphic systems. They are considered to be a form of Resistive RAM and were initially theorised in Chua (1971), but a physical example was not produced until the work of Strukov, Snider, Stewart and Williams (2008). A memristor's electrical resistance is not constant (as a typical passive resistor) but depends on the immediate history of current flow through the device. They therefore have a 'memory'; *i.e.* they are non-volatile, as when their current supply is removed, the memristor will remember its most recent resistance until current is applied again (Chua, 2011).

They are, therefore, perfectly suited for representing bistable synapses in neuromorphic VLSI ASIC devices. Typically, synapse weighting in devices such as that introduced in Moradi and Indiveri (2013) are implemented with transistor gates. This forms a volatile memory. When power is removed from the device, the device's synaptic state is lost. More importantly for low power contexts, this means that current must be constantly applied when the device is in use, even when it is idle. The use of a memristor in place of this traditional synaptic memory structure could theoretically lower the current draw of such devices. By shrinking the die area required for a synapse due to needing only one component rather than several, it will potentially increase the density of neurons and synapses on the same die (Gelencsér, Prodromakis, Toumazou & Roska, 2012). Such devices can be designed to implement STDP- or SDSP-type learning (Zamarreño-Ramos et al., 2011). Unfortunately, current memristive devices suffer from relatively high device mismatch, and as a result may introduce undesirable characteristics in synaptic dynamics (M. Hu, Li, Chen, Wang & Pino, 2011).

A crossbar-style synapse array using memristors was introduced in Jo et al. (2010), where its ability to represent the canonical STDP curve was demonstrated. Indiveri,

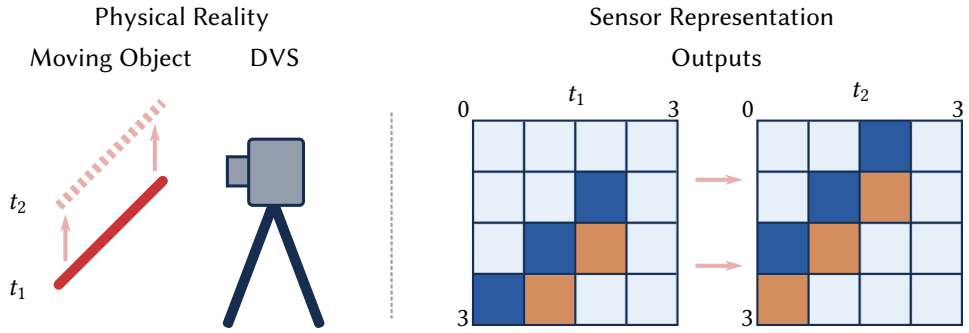


Figure 7.5: Block diagram of a simplified (4x4 pixel) Dynamic Vision Sensor responding to a moving stimulus. From left to right, we see: the stimulus (diagonally oriented bar moving upwards), the DVS, and its AER based responses to the movement at times t_1 and t_2. Dark blue pixels represent positive (excitatory) spikes, orange pixels represent negative (inhibitory) spikes. Although this figure has been presented with fixed timesteps, the DVS itself is asynchronous and timesteps are only imposed when recording input on a digital computer.

Linares-Barranco, Legenstein, Deligeorgis and Prodromakis (2013) introduce a more effective memristive synapse architecture, which uses different scales of memristors to emulate realistic biophysical structures. They note in this paper that memristive devices are uniquely suited to the simulation of *spiking* neural networks, and are difficult to implement for traditional real-valued neural networks. The interested reader to the review in Serafino and Zaghloul (2013) or Indiveri et al. (2013) for further discussion of memristors.

7.1.5 Applied Neuromorphic Hardware Systems

As the underlying concepts of neuromorphic hardware mature, a number of applied systems have become viable. Of particular interest (given the context of this thesis) are the Dynamic Vision Sensor and the general concept of neuromorphic prosthetics control. These will be briefly introduced here.

7.1.5.1 Dynamic Vision Sensor

The Dynamic Vision Sensor (DVS) is a biologically-inspired vision sensor whose pixels respond asynchronously to relative changes in intensity, much as the human eye does. Introduced in Lichtsteiner et al. (2006) and improved in Lichtsteiner et al. (2008) and subsequently in Yang, Liu and Delbruck (2015), the sensor output is an asynchronous stream of pixel AERs that directly encode scene intensity changes

at each pixel. See Figure 7.5 for a visual intuition of this. The DVS is also capable of a much higher dynamic range than normal Complementary Metal-Oxide Semiconductor (CMOS) cameras, closer to the capacity of the human eye. These properties are achieved by modelling three key properties of biological vision: its sparse, event-based output; its representation of relative luminance change (thus directly encoding scene reflectance change); and its rectification of positive and negative signals into separate output channels. The DVS asynchronously responds to temporal contrast rather than absolute illumination. The pixel response is based on a logarithmic intensity difference (either positive or negative) at that pixel, and is formulated as

$$\frac{\mathrm{d}}{\mathrm{d}t}\log I = \frac{\mathrm{d}I/\mathrm{d}t}{I} \tag{7.1}$$

The DVS has pixels that produce an 'on' (+) or 'off' (-) event signifying quantised increases and decreases of log intensity since the last event from the pixel. This is analogous to the functionality of a human eye's rod photoreceptor, which is responsive only to changes in relative intensity. It has already been used successfully for a number of applications, including high-speed robotic target tracking (Delbruck & Lichtsteiner, 2007) and traffic data analysis (Litzenberger et al., 2006).

In Figure 7.5, we would see that the AER output would be something like

$$\mathrm{AER}_{\mathrm{out}}(t_1) = \begin{cases} + = \{(0,3),(1,2),(2,1)\} \\ - = \{(1,3),(2,2)\} \end{cases} \tag{7.2}$$

$$\mathrm{AER}_{\mathrm{out}}(t_2) = \begin{cases} + = \{(0,2),(1,1),(2,2)\} \\ - = \{(0,3),(1,2),(2,1)\} \end{cases} \tag{7.3}$$

The time resolution has also been simplified in this example. In reality from a typical DVS we would receive spikes at a 50 µs rate. This is of course with the obvious caveat that these spikes are asynchronously processed, and time windowing is only applied by the microcontroller handling the input-output processes required for USB interfaces with commodity computers. These sensors can be truly asynchronous when connected to neuromorphic hardware systems like the INI VLSI chips, which communicate through AER (*cf.* Section 6.3.3.1). An example of asynchronous visual data processing using the DVS and INI neuromorphic chips is given in Indiveri et al. (2015).

7.1.5.2 Neural Prosthetics

MNPs, more commonly known as 'neural prosthetics', have the potential to help restore motor functionality for patients suffering from a wide range of neurological injuries and disorders. These systems convert the electrical activity measured from neurons into control signals, which can then be used to guide assistive technologies, such as prosthetic arms or computer pointing devices (Chestek & Shenoy, 2012). In this way they are a specific implementation of a BCI. The electrical activity of neurons can be acquired from a variety of electrodes. These signals are acquired using custom electronics. Neural decoder algorithms are then applied to this neural activity to classify it, and in most cases, approximate movement commands for the physical prosthetic device the neural prosthetic is controlling.

Neural prosthetics are inspired by a large body of prior literature in basic motor control neuroscience that identified the relationship of the firing rates of individual neurons to various movement parameters such as position, velocity, and force (Evarts, 1968; Georgopoulos, Kalaska, Caminiti & Massey, 1982). The first human experiment using an implantable neuroprosthetic device was described in Kennedy and Bakay (1998), where electrodes were implanted in a patient who had lost the ability to communicate due to amyotrophic lateral sclerosis. The patient was able to voluntarily modulate the recorded signals. The first two primate experiments demonstrating real-time control of a computer cursor by an ensemble of neurons in D. M. Taylor, Tillery and Schwartz (2002) and Serruya, Hatsopoulos, Paninski, Fellows and Donoghue (2002). Brain Machine Interfaces are the focus of a rapidly growing field, with a wide range of applications including localisation of seizures in cortex (Leuthardt, Schalk, Wolpaw, Ojemann & Moran, 2004), and wheelchair or prosthetic limb control (Hochberg et al., 2006; S.-P. Kim, Simeral, Hochberg, Donoghue & Black, 2008). Recently, attempts have been made to address the oversimplified notion of a linear relationship between neural activity and end point position or velocity. Non-linear machine learning techniques such as classical neural networks have been used (Aggarwal et al., 2008). There is also a movement towards using more biologically plausible activity decoders (Fagg et al., 2007; Pohlmeyer, Solla, Perreault & Miller, 2007). This is a potential opportunity to include spiking neural networks, as they are effective for use on spatio-temporal sequences with non-linear relationships. Appendix A will introduce an example case study on the classification of neuroinformatics data which shows the NeuCube is indeed applicable in this domain.

7.2 NeuCube on the Zhejiang FPGA

Zhejiang University in Hangzhou, China, have recently initiated the development of a neuromorphic simulation system, based on commercial off-the-shelf FPGA. This system is still very much in its infancy, and has yet to be formally published or released. The following details were released in personal communication with the project leader, Z. Gu (August 2015). Here, we introduce the preliminary steps of a project to implement the NeuCube on this hardware system.

The system itself provides the capacity for 2048 neurons, with full connectivity over a 16-bit synapse resolution. Axonal delays (*i.e.*, axonal distances) are supported up to a duration of 16 time units. These time units are arbitrary and functionally are tied to the chip's clock rate, but can be generally assumed to represent 1 ms per unit. Interestingly, this system supports models defined in PyNN (*cf.* Section 3.8.3), so in theory the NeuCube version introduced in Section 6.4 can be applied directly to this hardware.

The axonal delay constraints should not impose any significant restriction on small-scale NeuCube reservoirs, as they are to some extent shared by the SpiNNaker device. See Section 7.4.3.1 for a mitigation strategy developed for the SpiNNaker, which would be equally applicable here.

Internally, the models defined in PyNN are converted by the FPGA's support softare to a basic matrix of connection weights and delays. This matrix already exists in the JSON-formatted model save files we can generate with the M1 or PyNN implementations of the NeuCube. It is therefore possible, in the case that we have a pre-trained reservoir, for us to skip the model definition in PyNN for this particular device, and simply load these connection matrices.

Unfortunately, this system does not yet support any form of synaptic plasticity (*e.g.* STDP or SDSP). We therefore cannot train a NeuCube reservoir or classifier on this hardware device. This limits the device to acting as a 'readback'; *i.e.* no evolution of the reservoir is possible when it has been translated to this hardware. At this point, it is fixed, and cannot adapt further to changing stimuli. In certain contexts, this is not neccesarily a catastrophic issue. If we were to utilise the system in some environment where we had sufficent inital training data, and knew that the data context in which it would be applied was relatively static, a system could be trained in software and deployed onto the FPGA for power savings. For example, consider computerised fruit quality indication on a mechanised packing line; that set of input data (*e.g.*

individual fruit weights, visual data) would be relatively static and unlikely to require further on-line adaptation of the system. In the case adaptation was required, a model could be retrained offline and then redeployed. This is in contrast to the typical usage case of the NeuCube system, where online evolution of classification ability is required. Synaptic plasticity (STDP) is planned for this device, but as yet has not been implemented.

While at present no empirical evidence of this system's effectiveness is available, experiments are planned in the future to show the NeuCube's effectiveness when applied to such a computational platform. Firstly, we must show that the FPGA based system behaves acceptably when compared to reference NeuCube implementation (here, we have chose the M1). To test this, we must develop a version of the reference NeuCube which fixes its synaptic weights at the end of the initial training phase – *i.e.* STDP is replaced with static synapses with those end weights. A network will be trained to this state, then a copy exported to the FPGA system. Both networks (M1-fixed and the FPGA) will be run on the same data and their behaviours compared. Subsequent to this, a version of the NeuCube with applications for real-time robotics control will be generated and trialled on this hardware device.

This device is a good opportunity to show the NeuCube's robustness when it is applied to an environment where it cannot adapt further, and its applicability on FPGA-based neuromorphic hardware. Additionally, it is a useful proof of the Zhejiang system's functionality, and can assist in the validation that both systems behave as expected.

7.3 NeuCube on the INI Neuromorphic VLSI

The concept of neuromorphic VLSI ASICs was briefly introduced in Section 7.1.2. Here, we discuss the considerations for implementing the NeuCube framework on a specific group of devices currently in development at the INI, a joint research group within the Eidgenössische Technische Hochschule Zürich (ETH Zürich) and University of Zürich. Two such devices will be discussed here; the 256 neuron Reconfigurable On-Line Learning Spiking (ROLLS) system presented in Qiao et al. (2015), and the 9,000 neuron cxQuad system more recently introduced in Indiveri et al. (2015).

Initially, it was intended that one of the contributions of this thesis would be a NeuCube implementation on these devices. Unfortunately, due to the extended timelines required for the other contributions and the changes to the platform available, it

was not feasible to complete this implementation. Instead, here we will discuss the NeuCube on these devices in the context of a a future development. The contributions of this section, then, will be primarily theoretical, including a discussion of the considerations for implementing NeuCube on these hardware devices. Firstly, we identify the properties of the ROLLS and cxQuad systems, their HDL PyNCS, and the considerations for implementing NeuCube on them. As our primary interest is in the cxQuad system for its scale, the ROLLS will be introduced briefly for context.

7.3.1 Chip Architecture

The ROLLS and cxQuad chips are fabricated using a 180 nm 1P6M CMOS process (Indiveri et al., 2015). Smaller fabrication processes are possible, but have the effect of increasing device mismatch – *i.e.* the variabiliy of otherwise fixed parameters such as firing threshold and membrane leak rates – along with both design and production cost. This is particularly true when dealing with sub-threshold analog spiking neurons, as their dynamics are more directly affected by component variability than above-threshold digital systems.

Both of these devices use the Adaptive Exponential Integrate-And-Fire (AdEx) model neuron introduced in Brette and Gerstner (2005) and discussed in this thesis in Section 3.3.7. The model type is a physical property of the circuit, and cannot be changed. The parameters of these neurons, however, can be changed within some constraints; most particularly, that they must be homogenous across the chip (*i.e.* all neurons share the same parameters), and are altered within the constraints of the chip biases discussed later in this section. The schematic of the AdEx neuron's implementation on both of these chips is given in Figure 7.6. These devices work in real-time, and are continuously operating when power is applied.

Despite these similarities, the devices in question here differ in a number of key areas. Here, we briefly highlight the features which will most significantly affect our implementation of the NeuCube. The interested reader is directed to the introduction of the ROLLS in Qiao et al. (2015) or the cxQuad in Indiveri et al. (2015) for more comprehensive and technical discussion of these architectures.

7.3.1.1 ROLLS Architecture

The Reconfigurable On-Line Learning Spiking (ROLLS) device comprises 256 analog AdEx neurons and 133,120 bistable plastic synapses (Qiao et al., 2015). Three synapse types are implemented in this system, divided between 256 linear time-multiplexed,

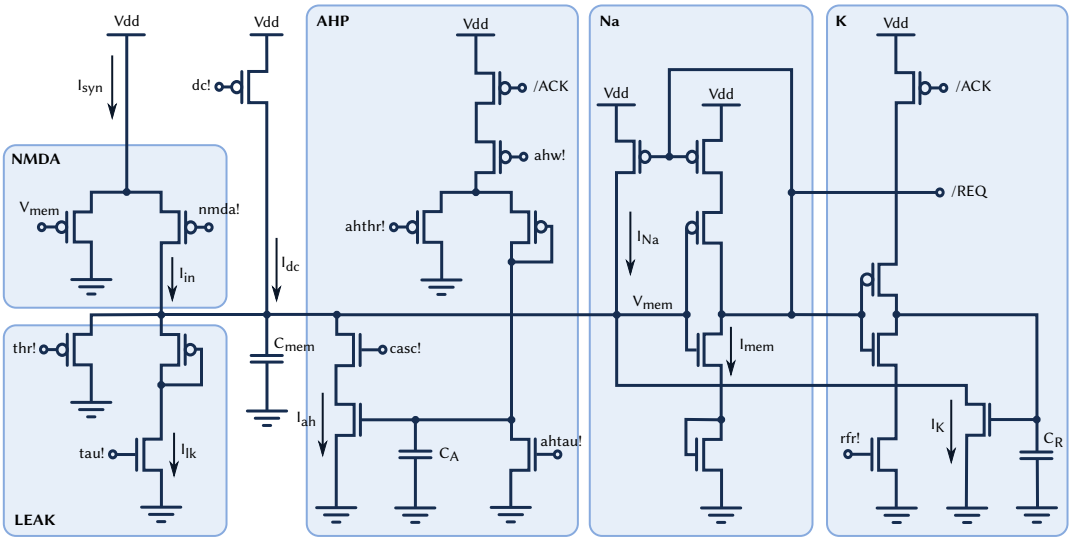


Figure 7.6: Simplified schematic diagram of the cxQuad and ROLLS chip neuron circuits. Input currents produced by the synapses I_{syn} are injected into the neuron membrane capacitance C_M, in parallel with a programmable constant DC current. The NMDA block models the voltage-gating mechanisms of NMDA synapses. The LEAK block models the neuron's leak conductance. The AHP block models the generation of the after hyper-polarizing current in real neurons, responsible for their spike-frequency adaptation behavior. The Na and K blocks model the effect of Sodium and Potassium channels respectively. REQ and ACK signals represent the digital voltages used to communicate Address-Events to the output AER circuits. All signals ending with '!' represent global neuron firing variables. The I_{mem} and I_{ah} currents represent the fast and slow variables in the AdEx model, respectively. Figure and caption reproduced from Indiveri, Corradi and Qiao (2015).

64,000 short-term plastic and 64,000 long-term potentiated. Learning in these synapses is consistent with the SDSP model introduced in Fusi (2000) and discussed in this thesis in Section 3.6.1.2. This system is of interest despite its limited number of neurons as it is an ideal candidate for the implementation of the deSNN learning algorithm discussed in Section 3.6.3.2.

7.3.1.2 cxQuad Architecture

The following characterisation is drawn from Indiveri et al. (2015) and discussion with the developers (G. Indiveri & N. Qiao, personal communication, April 2015). A hybrid digital-analog system, the cxQuad chip comprises 1,024 neurons with around 64,000 Content Addressable Memory (CAM) programmable synapses divided over four cores. The synapse and neuronal dynamics calculations are implemented in analog hardware, with an asynchronous digital communication infrastructure developed to service the AER-based spike transmissions. This communication fabric extends

across cores and beyond the chip edge. In this way, a number of these chips can be joined to scale the device in a modular fashion. Spikes are sent in a multicast fashion to a maximum of four cores over these digital links, and should pass from chip to chip in around 15.4 ns. Due to this communication structure, a single neuron can potentially address up to 1,024 neurons with each postsynaptic spike. However, as discussed below, each neuron can only receive from 64 presynaptic neurons.

Each chip contains a 16×16 matrix of neuron 'units', the totality of which is referred to here as a 'core'. Each of these contain a single neuron, and a block of 64 incoming synapses. These synapses can be further subdivided into groups representing fast and slow excitation and inhibition through the use of the on-chip bias generators, which are controlled by the user. Each of these synapses is associated with an address in the AER protocol used, and can only be addressed by one neuron. Therefore, each neuron can receive presynaptic spikes from a maximum of 64 neurons.

Previously, it was mentioned that the neuron parameters on these devices must be homogenous; in fact, in the cxQuad this constraint applies only across the *core* (*i.e.* this 256 neuron matrix), not the entire chip. It is therefore possible to have four populations with distinct properties across each cxQuad chip.

Perhaps the most interesting device at present in the context of the NeuCube project combines nine of these cxQuad chips onto one board, providing a total of 9,096 neurons and 5,821,440 programmable synapses on the same device. It is this board which we intend to utilise in the future, as it provides sufficent scale for a typical NeuCube reservoir at significantly reduced power cost.

7.3.2 PyNCS

PyNCS[1] is a Hardware Description Language (HDL) interface introduced in Stefanini, Neftci, Sheik and Indiveri (2014) for the definition of neural network models to be run on certain neuromorphic ASICs, primarily those developed at the INI. The syntax of PyNCS is intended to be similar to that of the Brian simulator, with added considerations for the specific requirements of hardware setup. It may in the future be possible to integrate PyNCS with PyNN, extending the PyNN interface with the capability to be run on neuromorphic ASIC devices. This concept is addressed in Stefanini et al., where it is suggested that while the two interfaces share some common characteristics, their design philosophies are divergent. In any case, the design patterns used in the development of the PyNN version of the NeuCube should

[1] https://github.com/inincs/pyNCS or http://inincs.github.io/pyNCS/

map to the PyNCS version with minimal changes, even without this PyNN–PyNCS integration.

Both the ROLLS and cxQuad devices support models defined in PyNCS. However, as the cxQuad system is so new (indeed, at the time of writing the introductory paper had not yet been presented), a number of other support systems are still in development. In particular, software to manage the mapping of neurons to chip locations is not yet released. Without this software, it is not feasible to effectively partition and place the network components and their respective routing tables without intimate knowledge of the specific chip architecture. This functionality is currently in production, and once it is released, we intend to implement the NeuCube here. Realistically, the development of the PyNCS code should be relatively straightforward once this mapping system is completed. The challenge to effectively implement the NeuCube on these devices is in the tuning of the chip biases. This issue is discussed in the next section.

7.3.3 Considerations for the NeuCube on INI Neuromorphic VLSI

The most obvious initial consideration for the implementation of the NeuCube on such a device is the constraints on synapses. Typically in a computational simulation or even in the case of some digital neuromorphic hardware (*e.g.* the SpiNNaker) we have no meaningful constraint on the number and type of synapses used, save for the tedium of model construction times. In the case of these ASICs however, there are physical constraints on the number and behaviour of these synapses.

Most particularly, in the case of the cxQuad they are one of four types; fast or slow excitation or inhibition. The synapse type is set using two of the 25 biases each core can be controlled by. These synapses do not have a 'weight' as in traditional synapses – their dynamics are controlled entirely by their bias setting. Learning can therefore be implemented in a limited manner by altering the synapse type. This would certainly result in a different network behaviour to an otherwise identical network modelled in software, as the implementation of the unsupervised learning phase of the NeuCube would therefore be different. We should be aware that these network dynamics would be different, possibly leading to different training and classification behaviour. This issue would make it difficult to directly map a pre-trained model from or to this device, in much the same way as is discussed in Section 6.4.4.

Synapses are also limited to 64 incoming per neuron. This is a physical property of the chip, and cannot be changed. Smaller numbers of synapses may be used, but

this is the upper bound. Whether this is sufficent depends entirely on the reservoir implemented and the particular data context in which the system will operate. Recall that a reservoir's pattern separability and memory increase with larger numbers of synapses. It may be that 64 incoming synapses is insufficient for a particular type of highly-complex data; unfortunately, it is difficult to ascertain ahead of time whether this is the case, as is discussed in Section 5.2. Recall that a typical human neuron has on average 1,000 incoming synapses. In the case that we were intending to perform a simulation of this type of network, the choice of this particular hardware device would therefore likely be sub-optimal. The minimum number of synapses a NeuCube has been run with at present is 100; it is a necessary future work to verify the NeuCube's behaviour and performance under this constraint.

Similarly, there is a structural constraint on the number and type of neuron used. In a computational simulation we have complete freedom to use any neuron model we choose, provided that we have sufficent computational power to support it. In this case, our choice of neuron is fixed by the nature of the circuit which models it. We must show that the NeuCube can function in a satisfactory manner using the AdEx neural model prior to its application on this device.

Additionally, we must show that the NeuCube can function with a limited number of these neurons. This second issue is less pressing; contemporary applications of the NeuCube have used a maximum of around 5,000 neurons, and this framework shows excellent results with significantly fewer (*cf.* Appendix A). As the scale of the NeuCube grows, this may be a consideration in the future.

Due to the fact that these devices operate in continuous real-time, they are ideally suited for application in areas such as BCI or robotics control. The formulation of the NeuCube framework does theoretically support continuous-time operation of both the reservoir. Some output devices including the SPAN classification algorithm also support a continuous formulation, and as such, it is feasible that the NeuCube framework could be implemented in this context in the future. Additionally, the extremely low power draw of these devices compared to even the SpiNNaker or Zhejiang FPGA systems (on the order of 945 μW in a worst-case) is ideally suited for application in low-power environments such as prosthetics control.

In order for us to exploit these properties, we must carefully select both our NeuCube framework components, and our application context. For example, it is likely meaningless to model a financial time series in such a system. However, the application of a BCI is admirably suited. Applications of the NeuCube on these devices should

primarily be chosen to take advantage of the key benefits of such systems; extremely low power consumption, continuous- and real-time operation, and interoperability with AER-based sensor or communication systems. Furthermore, as stated, it is necessary to show that the NeuCube is indeed still effective under the constraints that such systems impose. To this end, a software simulation of the NeuCube using the AdEx will be implemented, and comprehensively explored. This implementation will also explore the effectiveness of the defined synapse dynamics. In doing so, we should establish a baseline for expected network behaviour which can be used in tuning the chip biases, and ensuring that the network behaviours are meaningful and reproducible.

The applicability of the NeuCube framework to such a computational platform is obvious. Taking as given that the above experiments are satisfactory, in the near future a NeuCube implementation for the cxQuad system will be developed and applied to different contexts, including for real-time adaptive BCI.

7.4 NeuCube on the SpiNNaker

One of the most signficant contributions of this thesis is a massively-scalable, high-performance implementation of the NeuCube on an emerging neuromorphic hardware platform: the SpiNNaker. Using this system, NeuCube reservoirs can now be scaled up to arbitrarily large numbers of neurons and synapses. Additional to the scaling, the same network simulated on a SpiNNaker device will utilise an order of magnitude less electrical power than a standard commodity computing simulation, and run at a guaranteed constant wall-clock speed. By this, it is meant that a simulation with 1,000 neurons or 100,000 neurons run for the same simulated time will take the same real-world (wall-clock) time to complete, regardless of their network size.

This ability is necessary going forward with the NeuCube. As the problem domains in which it is applied become more complex, and the data volume we deal with becomes larger, it is necessary to scale the model up in terms of reservoir size. It is a well established property of SNN that memory capacity increases with the number of synapses, and so too does the network's non-linear pattern separability (Kasabov et al., 2005b). Correspondingly, we therefore need a scalable computational platform. Our motivation for utilising neuromorphic systems in general to solve this issue was established at the beginning of this chapter. Our motivation for utilising the SpiNNaker specifically, will be explored in this section. Its primary advantages for the NeuCube are that it provides an arbitrarily scalable, low power, adaptable

computational platform which is relatively straightforward to develop for. This last factor is, in the context of the NeuCube, its strongest benefit when compared to the ASIC and FPGA systems introduced in Sections 7.3 and 7.2 respectively.

A brief discussion of the feasibility of implementing the NeuCube on Spinnaker was previously published in Scott and Kasabov (2015), and in Kasabov et al. (2015).

Here, the SpiNNaker device and its place in the current literature is explained. Secondly, in Section 7.4.2, some of the most significant considerations for implementing the NeuCube on this hardware, and how this may affect more complex networks as we iterate on the basic NeuCube framework are discussed. In Section 7.4.3, necessary changes to the PyNN version of the NeuCube introduced in Section 6.4 are defined, in order to run it on the SpiNNaker.

7.4.1 The SpiNNaker Device

SpiNNaker is a general-purpose, scalable, multichip multicore platform for the real-time massively parallel simulation of large scale SNN, developed as part of the European Union Human Brain Project (Furber, 2012; Furber et al., 2013; Rast et al., 2010). Each SpiNNaker chip contains 18 subsystems ('cores' here) responsible for modelling up to one thousand neurons per core. The architecture of these cores can been seen in Figure 7.7, while the architecture of the chips themselves is given in Figure 7.8. These chips are connected to each other using six asynchronous links per chip, arranged hexagonally. For an intuition of this connection structure, see Figure 7.9. Spikes are propagated using a multicast routing scheme through packet-switched links. SpiNNaker is a Globally Asynchronous Locally Synchronous (GALS) system with processor nodes residing in what are termed 'synchronous islands', surrounded by the light-weight, packet-switched asynchronous communications infrastructure. Two SpiNNaker versions are available for the work in this thesis; a small 4-node board, and a full-size 48-node board which are subsequently referred to as the SpiNN-3 and SpiNN-5 types respectively. The differences between these versions are explained in more detail later in this section.

The philosophy behind the system assumes that processors are 'free': the real cost of computing is energy. In fact, this is borne out in commodity computing systems as well, as the lifetime cost of power to run a commodity supercomputer is typically one to two orders of magnitude higher than the inital cost of the hardware (Feng, 2003; Storlie et al., 2014). Low power ARM968 cores have been utilised to greatly enhance power efficiency at the cost of some performance (Furber et al., 2013).

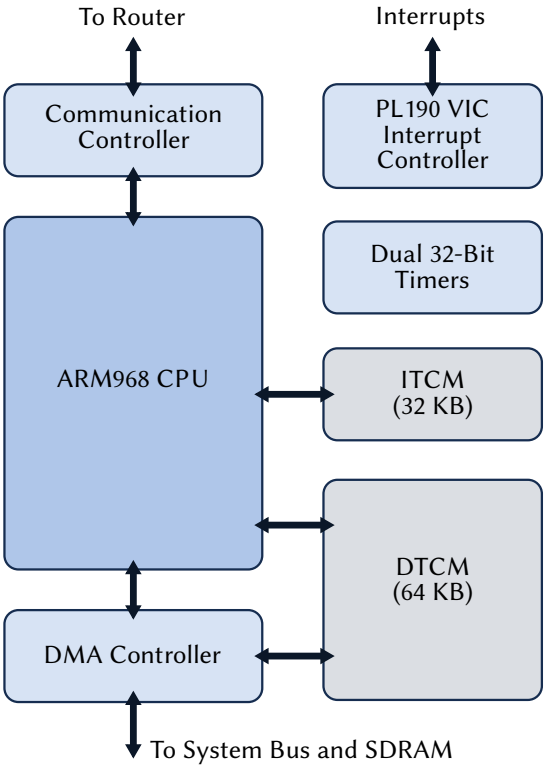


Figure 7.7: Block diagram of the architecture of a single SpiNNaker core. Each SpiNNaker chip contains 18 of these. Each core has 32 KB of Instruction Tightly-Coupled Memory and 64 KB of Data Tightly-Coupled Memory, and a dedicated Direct Memory Access controller for both these and the whole-chip SRAM. These custom components surround a standard 200 MHz ARM968 processor.

The physical hierarchy of the system has each node containing two silicon dies: the SpiNNaker chip itself, and the SDRAM, which is physically mounted on top of the SpiNNaker die and wire-bonded to it. The nodes are packaged and mounted in a hexagonal array (Figure 7.9) on a PCB. The full system requires 1,200 such boards. In operation, the full SpiNNaker will consume at most 90 kW of electrical power (Furber et al., 2013; Sharp, Galluppi, Rast & Furber, 2012). The maximum draw for a single SpiNN-3 is 5 W, and for a single SpiNN-5 around 72 W. This is advantageous when compared to the same network run on a commodity computer which would typically be on the order of 400-600 W. In simulation, Sharp et al. report the SpiNN-5 consumes 100 nano-Joules per neuron per millisecond and 43 nJ per postsynaptic potential, the smallest energy consumption reported for any digital computer simulation of SNN. Of course, the subthreshold ASICs of Indiveri et al. are on the order of pico-Joules per postsynaptic potential, but with the consideration that they are less adapable.

The system typically runs in real-time (*i.e.* that 1 ms wall clock time = 1 ms simulation time), which means that it is feasible to implement interactive applications that

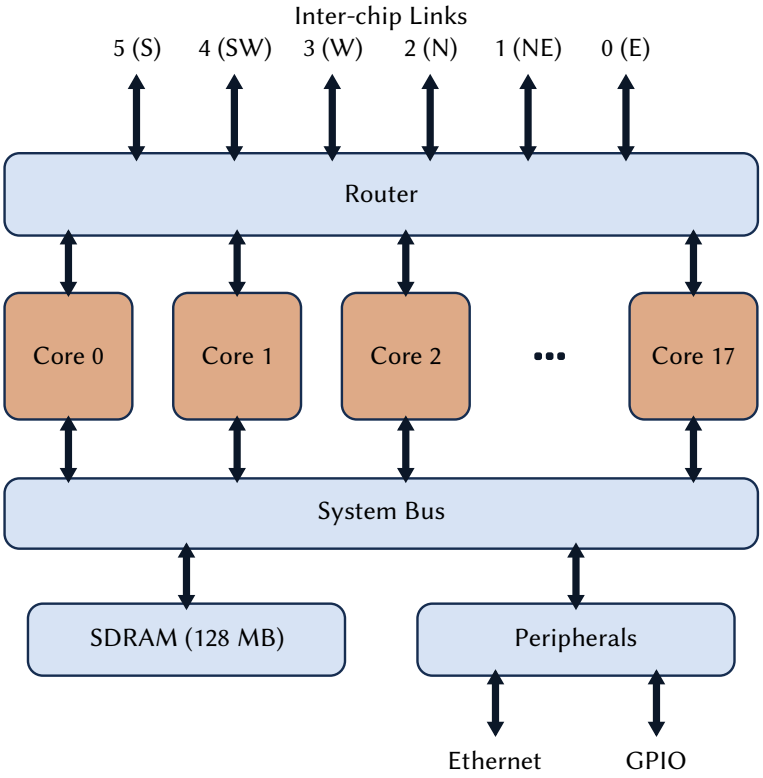


Figure 7.8: Block diagram of the architecture of a single SpiNNaker chip. Each chip contains 18 SpiNNaker cores (*cf.* Figure 7.7), a whole-chip router, and system bus to the peripherals and shared 128 MB SDRAM.

can inject spikes or change synaptic weights online (Galluppi et al., 2010). This is particularly useful for the NeuCube, as it means that we can implement the learning rules in a generic model and run this on the SpiNNaker hardware in real-time. The architecture is also such that it performs best when neural connections exhibit some kind of locality (Jin, Rast, Galluppi, Davies & Furber, 2010). It is noted in that same paper that distance-dependent connectivity is optimal for the SpiNNaker system. Our small-world lambda-distance connectivity is perfectly suited to this architectural constraint. In addition, even though the chips are connected hexagonally, a standard Cartesian coordinate system is suitable for these connections (Khan et al., 2008). The SpiNNaker also implements a form of STDP between its connections (Jin et al., 2010), further reinforcing its suitability for a platform to implement NeuCube on.

Applications can be programmed for SpiNNaker using a version of the high-level neural modelling description language known as PyNN, which has been discussed prior in this thesis (*cf.* Section 3.8.3). The SpiNNaker interface of PyNN first introduced in Galluppi et al. (2010) is now known as sPyNNaker, and is in fact a reimplementation of PyNN in the context of the SpiNNaker. See Section 7.4.2 for

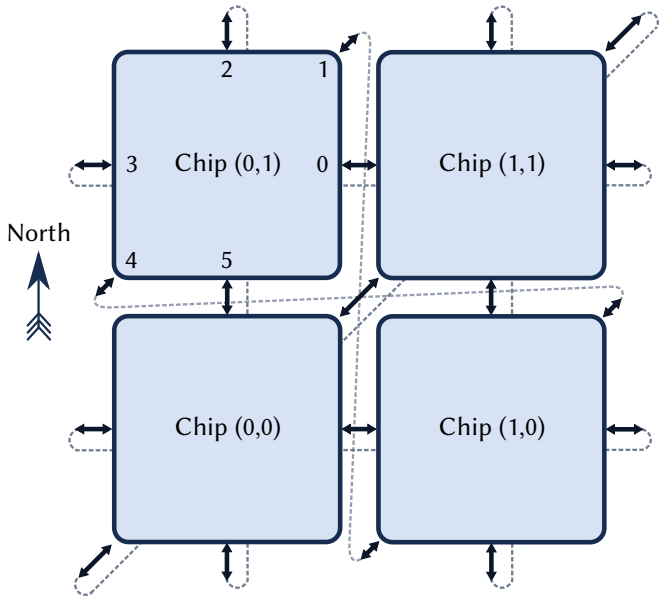


Figure 7.9: Block diagram of the SpiNNaker chip interconnection fabric. Each block represents one SpiNNaker chip (*cf.* Figure 7.8). The conceptual 'North' of the chip orientation is noted. Dotted lines represent conceptual 'wrap-around' links; *i.e.* the East-most connection of chip (1,1) is connected to the West-most connection of chip (0,1). We label the hexagonal interchip links anticlockwise from East. These numbers signify the link number identified in Figure 7.8. Chips are addressed using their Cartesian coordinates from bottom left. Using this connection fabric, an arbitrary number of chips can be connected together in a relatively simple way.

further discussion of the sPyNNaker software and challenges developing for it. It is intended that a generic model description written in PyNN (sPyNNaker) can be implemented transparently (to the user) on the SpiNNaker hardware just as it would be on a commodity desktop, although this is not necessarily the case when dealing with more complex models like the NeuCube. Alterations to the original PyNN code introduced in Section 6.4 in order to run it efficiently on the SpiNNaker are discussed in Section 7.4.3.

At start-up, the cores each bid to be elected to a special role as a Monitor Core and to thereafter perform system management tasks. In a typical case, 16 cores per chip are used for the actual simulation, and the remaining core is reserved as a spare for fault tolerance and to account for manufacturing yield problems. As previously mentioned, inter-processor communication is based on a multicast infrastructure. These packets are source-routed; *i.e.* they only carry information about the issuer and the network infrastructure is responsible for delivering them to their destinations (Furber et al., 2013).

In the SpiNNaker context, a network is treated as a graph; neurons are vertices, and synapses are edges. These are futher decomposed into sub-vertices and sub-edges when generating the machine-level representation of the network. These sub-vertices are broken down so as to be handled by a single core. This is termed *partitioning*. Each sub-vertex is allocated to a core on the SpiNNaker, known as *placement*. The edges of the graph – here, representing synapses – between these sub-vertices are translated into routing tables for the multicast 'spike' packets, thereby mediating neuronal communication. These are written directly to each chip.

The basic algorithm for a single simulation, therefore, is straightforward:

```
create network description
partition and place network graph
generate machine files
load files
run simulation:
  synchronise simulation start on all cores
  simulate network
  end simulation
readout results
cleanup and post-processing
```

Listing 7.1: Pseudocode algorithm of the control process of a single simulation run on SpiNNaker.

SpiNNaker is particularly interesting because of its hybrid nature; it draws some inspiration from more dedicated neuromorphic hardware systems, but all of the model and synaptic dynamics are performed in software. This means that it is theoretically an universal platform with respect to SNN simulation; no models are fixed in hardware (Khan et al., 2008). Interestingly, this not only means that it can handle networks of heterogeneous neurons of the same model type, but incorporate networks of heterogeneous model types in the same simulation. In fact, such an experiment was shown in Rast et al. (2011), where a network of both LIF and Izhikevich model neurons were simulated. This feature is of interest when applied to the NeuCube, as it means that different neuronal types can easily be incorporated into the same NeuCube model depending on the specific task, and the model can be adapted rapidly to the same. In the same way, we can also use the SpiNNaker device to model other neuromorphic hardware implementations; for example, with a network of AdEx neurons with binary synaptic weights, we can simulate (to some reasonable degree of behavioural accuracy) the dynamics of the same network implemented in the INI neuromorphic ASIC. This is also possible in software simulation on commodity hardware, of course, but implementing such a meta-simulation on the SpiNNaker

would allow us to model much larger networks of other neuromorphic hardware systems than would otherwise be feasible.

A rough estimation for the number of LIF neurons a single core on a single chip can manage in real time is 256 without STDP, and around 128 with STDP. This is contingent on a number of factors, including the complexity of the learning model and the number of incoming synapses. More synapses, and more compex synapses (*i.e.* those with learning) will reduce the maximum number of neurons per core proportionally. This limitation is due to the memory structures used at present in the software underlying SpiNNaker, and is intended to be improved to roughly 1,000 neurons per core in later iterations of the software. The maximum number of neurons per core was reported in Serrano-Gotarredona, Linares-Barranco, Galluppi, Plana and Furber (2015) where a convolutional neural network of traditional real-valued neurons was was developed with 2048 neurons per core. This is an exceptional case, as convolutional neural networks can been implemented to share synaptic memory, so the computational cost can be significantly lower than using the unique synapses seen in most neural network models.

At present, there are two SpiNNaker boards available to users. The '102 machine', here referred to as the SpiNN-3, is the 4-node circuit board shown in Figure 7.10. It contains 72 SpiNNaker cores over four chips, which will typically be deployed as 64 application cores, 4 Monitor Processors and 4 spare. These chips are capable of a current theoretical maximum network size of around 16,384 neurons simulated in real time. This requires a 5 V 1 A (5 W) supply, and can in theory be powered from some USB ports. The control and IO interface is a single 100 Mbps Ethernet connection. There is limited provision for connecting cards together with SpiNNaker links to form larger systems or to other hardware systems like the DVS.

The other board currently available is the 48-node board shown in Figure 7.11, here referred to as the SpiNN-5, which comprises 864 SpiNNaker cores over 48 chips. These are typically deployed as 768 application cores, 48 Monitor Processors and 48 spare cores. This board requires a 12 V 6 A (72 W) supply. These chips are capable of a current theoretical maximum network size of around 196,000 neurons simulated in real time. The control interface is two 100 Mbps Ethernet connections, one for the Board Management Processor and the second for the SpiNNaker array. There are options to use the six on-board 3.1 Gbps high-speed serial interfaces (using SATA cables, but not necessarily the SATA protocol) for IO. The IO limitation is currently an issue for non-trivial networks. As discussed in Section 7.4.2.2 and Section 7.4.3.2, model IO times are much longer than the actual simulation run times.

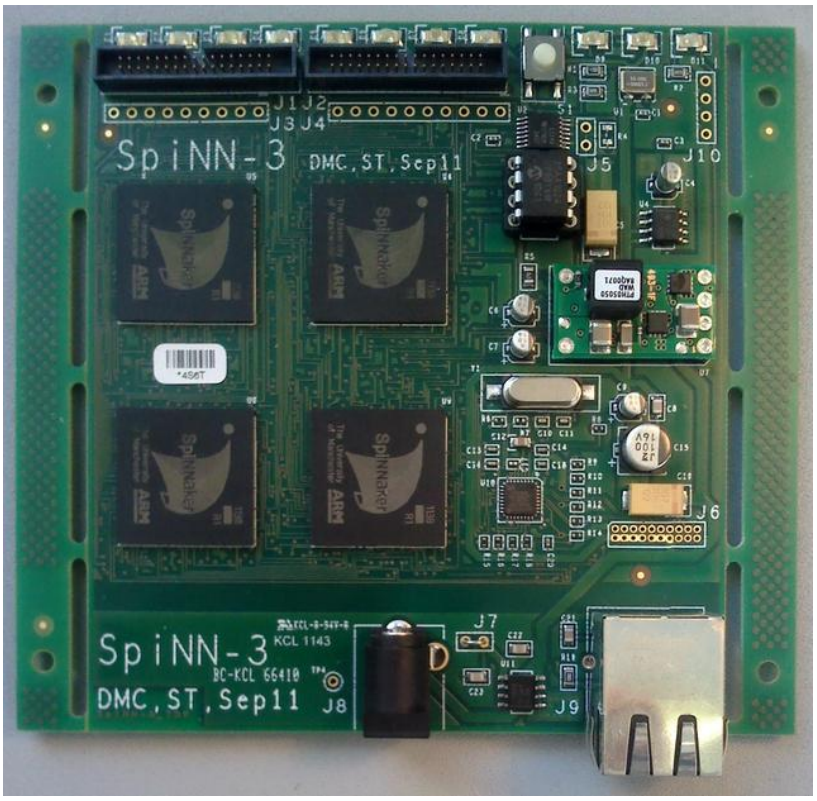


Figure 7.10: Photograph of a SpiNN-3 4-Node SpiNNaker board. The black chips in the centre are the four SpiNNaker chips each providing 18 SpiNNaker cores. A single Ethernet port are seen in the bottom left for board management and low speed IO.

These SpiNN-5 boards can be connected together to form larger systems using the SATA links. This is the basis for the construction of the eventual full configuration of SpiNNaker devices, modelling one billion neurons in real time at around 70 kW.

Early in this thesis, a SpiNN-3 board (Figure 7.10) was recieved on long-term loan from the APT group. This board presently is used for protoyping and testing of models prior to deployment on the larger SpiNN-5 board (Figure 7.11) KEDRI has subsequently purchased. In the following subsections, some of the challenges encountered when working with this device, and a number of specific behaviours for developers of NeuCube systems to consider will be discussed.

7.4.2 Significant Development Considerations

In developing for the SpiNNaker device, we must take a number of factors into consideration. While the SpiNNaker provides a number of advantages in terms of power cost, performance guarantees, and so on, our use of this device must be a mindful one; there is, unfortunately, ‘no free lunch’. Particular concerns when

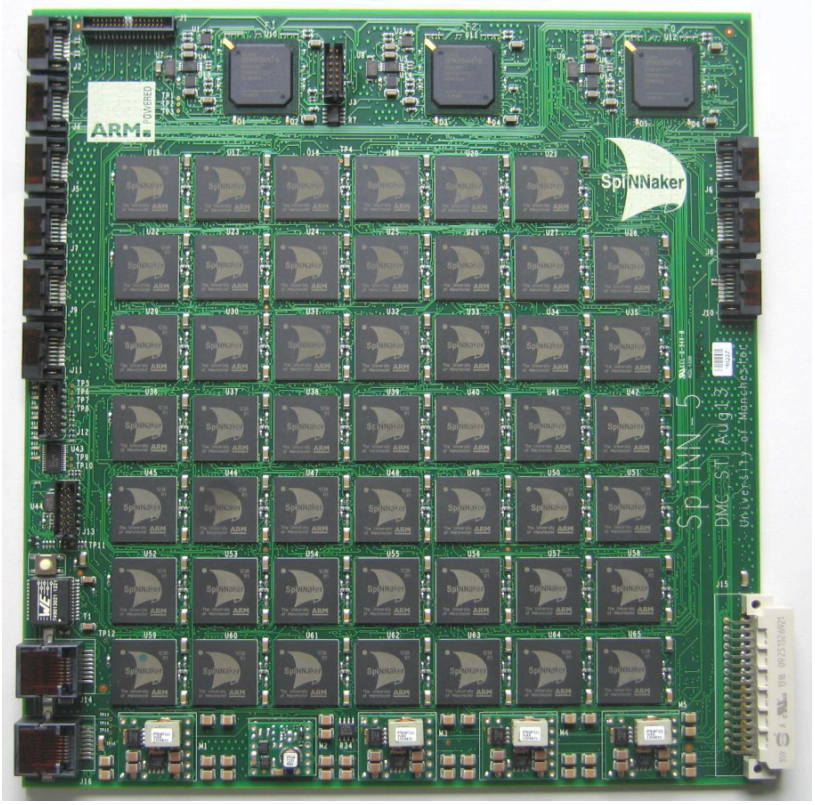


Figure 7.11: Photograph of a SpiNN-5 48-Node SpiNNaker board. The topmost three chips are Spartan-6 FPGA responsible for board IO management. The black chips in the centre are the 48 SpiNNaker chips each providing 18 SpiNNaker cores. Around the edges we see nine SATA ports for board interconnection and IO processes. Two Ethernet ports are seen in the bottom left for board management and low speed IO.

developing for this system can be separated into two primary areas; the relative immaturity of the system, and its architectural constraints.

By relative immaturity, we refer to those concerns discussed in Section 7.4.2.1, where it is made clear that the SpiNNaker is still very much in development. Some key features remain to be implemented due to the fact that it is a new simulation ecosystem, and that developer attention is currently being paid to their primary users – computational neuroscientists. However, it must be said that this system is developing rapidly.

By architectural constraints, those issues raised later in this section are meant, where the limited IO speeds, and the stochastic nature of simulations performed on it, are discussed, among others. Such constraints can affect the behaviour of a simulation run on this hardware, when compared to an identical simulation run in software. In addition, it may require structural changes to the simulation (*e.g.* mean spike rate, number of synapses per neuron).

Our decision to use the SpiNNaker device must be informed by these factors. Introduced here will be the most significant considerations towards implementing the NeuCube on a SpiNNaker device, and how these have affected the implementation introduced in this thesis in practice.

7.4.2.1 Code and Ecosystem Maturity of SpiNNaker

A key example of the relative immaturity of the SpiNNaker support ecosystem is given in Section 7.4.3.2, wherein a significant challenge faced when attempting to loop the same simulation with different inputs is discussed; remarkably, this is currently a non-trivial exercise in the SpiNNaker. Section 7.4.3.1 discusses an issue encountered when initially setting up a network, and the constraints on network definition in the existing software. In this secondary case, this issue is shared by both native PyNN and the sPyNNaker port.

Here, we will briefly describe here a representative example of one of the significant issues faced in developing the NeuCube for the SpiNNaker platform. This particular issue would cause the device to crash on initialisation of a network, when scaling high-resolution (32 bit floating-point) synaptic weights into the alternate resolution (32 bit fixed-point) supported by SpiNNaker.

In the generation of synaptic weights for a NeuCube reservoir, the current PyNN implementation randomly intialises these with a Gaussian distribution[2], the characteristics of which depend on the neuronal type (*cf.* Appendix C.3). For excitatory connections the distribution's $\mu = 0.5$ and $\sigma = 0.3$, and for inhibitory connections $\mu = 0.4$ and $\sigma = 0.2$. These weight distributions follow the generally accepted practice of excitatory connections being stronger than inhibitory ones.

It is theoretically possible for connection weights to be very close to zero, and outside of the range of synaptic weights that the SpiNNaker will support. In a general case, it is also possible for the synaptic weights to be significantly different in magnitude. In any case, weights are scaled during the simulation preparation phase of the SpiNNaker support software, with no intervention by the user. In the case that this weight scaling reduces some small values to zero, the system should automatically warn the user that this scaling has occurred. In the case of the early sPyNNaker code, this warning was implemented as an Exception. When this condition was reached, the Exception would be raised unhandled, and the system would crash.

[2]Using the common formulation of the Gaussian distribution, $p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

This particular behaviour is undesirable, and needed to be rectified in order to run a NeuCube reservoir on this system.

The issue described was identified as being caused by the Exception shown in Listing 7.2. Subsequent to consultation with the members of the SpiNNaker development group, it was determined that the changes shown in lines 101–104 of the same Listing were sufficient for a short term fix. This issue has subsequently been corrected permanently in the April 2015 release[3] of the sPyNNaker system. It now shows a non-halting warning, as desired.

```
78  def get_packed_plastic_region(self, synapse_row, weight_scales,
79                                              n_synapse_type_bits):
80      """
81      Gets the plastic region of the row as an array of 32-bit words
82      """
83      # Convert per-synapse type weight scales to numpy and
84      # Index this to obtain per-synapse weight scales.
85      weight_scales_npy = numpy.array(weight_scales, dtype="float")
86      synapse_weight_scales = weight_scales_npy[synapse_row.synapse_types]
87
88      # Scale weights
89      abs_weights = numpy.abs(synapse_row.weights)
90      abs_scaled_weights = numpy.rint(
91          abs_weights * synapse_weight_scales).astype("uint16")
92
93      # Check zeros
94      zero_float_weights = numpy.where(abs_weights == 0.0)[0]
95      zero_scaled_weights = numpy.where(abs_scaled_weights == 0)[0]
96      # ----------
97      # Removed by NS 03-MAY-15 at the suggestion of JK to correct
98      # for the issue where intialisation crashes when scaling
99      # weights - this should only be a warning, not an Exception
100     # ----------
101     #if (zero_float_weights.shape != zero_scaled_weights.shape or
102     #        (zero_float_weights != zero_scaled_weights).any()):
103     #    raise Exception(
104     #        "Weight scaling has reduced non-zero weights to zero")
105     # ----------
106     # Rest of method omitted
```

Listing 7.2: Alterations in SpiNNaker API file plastic_weight_synapse_row_io.py to ensure that synaptic weight scaling would no longer cause a failure of the SpiNNaker board boot sequence.

This state had not been reached prior in 'normal' operation, as the applications previously run on the SpiNNaker device had not required the specific connectome of NeuCube reservoir. Due to the relatively recent introduction and corresponding low level of maturity of the software ecosystem supporting the SpiNNaker, time was

[3] https://github.com/SpiNNakerManchester/sPyNNaker/releases/tag/2015.004

lost in the development of mitigation strategies for missing features. A number of key features required for the development of the NeuCube (including synaptic state readout) were not available until late in the development process of this thesis. For further examples, see the following Sections 7.4.3.1 or 7.4.3.2.

This software ecosystem is quickly improving; a number of the issues introduced here and in later sections are merely indicators of the fact that the SpiNNaker device is new and rapidly evolving. Additionally, the NeuCube is one of the few systems which applies the SpiNNaker in a context other than computational neuroscience. They are mentioned only due to their impact on the development process of the NeuCube, and are not intended to imply that the SpiNNaker system is somehow insufficient. In time, the issues raised here will be rectified.

7.4.2.2 IO Limitations

Currently, Input-Output (IO) speeds on the SpiNNaker device are limited. As previously discussed, IO between the device and the host computer is presently mediated by a 100 Mbps Ethernet link. In practice, the actual IO speed is around 30 Mbps, when processing on the SpiNNaker is taken into account. This is equally true for both the SpiNN-3 and SpiNN-5 boards. As a result, model saving and loading is slow (*cf.* Section 7.4.3.2) and high-speed input data streaming from a host computer is not realistic. It is possible to stream high speed, asynchronous spike data from a neuromorphic device compatibile with the FPGA headers available on the SpiNN-5.

This issue limits our possible areas of exploration for the NeuCube on this computational platform. For example, the application context introduced in Appendix B – where we approach the classification of very high speed radioastronomy data using the NeuCube on a neuromorphic system – is not feasible with such low IO speeds.

There are options to use the SpiNN-5's on-board 3.1 Gbps SATA interfaces for IO. Recall that these links are intended to be used in connecting the devices together to form the large toroidal network. However, only five of these links per device are technically necessary for this, leaving four free for general purpose IO. The software configuration of the on-board FPGA for these links to be used in this way does not yet exist, and in addition, they would most likely require some custom interface hardware for the host computer.

A custom solution for this host device interface through SATA, used for streaming DVS sensor data over one of the SpiNN-5's SATA links, has been developed by the

University of Seville. Introduced in Iakymchuk et al. (2014), this 'infrastructure card' device uses a RaggedStone[4] FPGA development board containing a Spartan-6 FPGA, identical to those used in the SpiNN-5 device. A custom interface card from whatever input device we desired would need to be built and linked to this infrastructure card, which is then responsible for the encoding of this information into the SpiNNaker AER format, and injecting these formatted packets into the SpiNN-5's on-board FPGA. Note that this encoding is not encoding into trains of spikes; that process would need to be implemented prior to interfacing with the infrastructure card. This is an interesting development, but unfortunately not immediately applicable to our work. After consultation with the developer (B. Linares-Barranco, personal communication, April, 2015) it is clear that this device is not feasible to be used by those without intimate knowledge of both the SpiNNaker hardware architecture and FPGA programming.

For the moment, the Ethernet link limitations will need to be accepted, until such time as the SATA links become available for general purpose IO.

7.4.2.3 STDP Implementation

In most desktop computer simulations, the implementation of STDP is quite straightforward. Because all synaptic weights are locally accessible, STDP can be calculated at spike emission or reception. However, this straightforward implementation is not the case with SpiNNaker. Recall from Section 7.4.1 that the SpiNNaker device uses a shared memory system, where the synaptic weights are stored externally on SDRAM, keyed to the presynaptic index and stored in the memory of the postsynaptic neuron. They are recalled from memory when a presynaptic spike arrives. In this structure, presynaptic neurons have no knowledge of their synaptic afferent on the postsynaptic neuron, and canonical STDP can therefore not be implemented as-is (Furber et al., 2013; Jin et al., 2010).

To address this, Jin et al. (2010) introduced the deferred event-driven (DED) model. In DED, the STDP calculations are deferred until there are sufficient spike timing records to calculate the pre- and post-synaptic weight changes, and the weight update is applied retroactively. The detailed algorithm is discussed in Jin et al., and will be omitted here for brevity. In this way, STDP can be applied even without foreknowledge of the synaptic weight or postsynaptic neuronal dynamics.

[4] http://www.enterpoint.co.uk/products/spartan-6-development-boards/raggedstone-2/

Alternative implementations of STDP have subsequently been proposed. Davies, Galluppi, Rast and Furber (2012) introduced a forecast-based form of STDP, which uses a new learning rule they have termed TTS (Time-To-Spike) for the LTP component, and standard STDP rule for the LTD component of the learning. This TTS method uses a statistical approach, with the basic concept that the higher a neuron's membrane potential, the higher the liklihood of it emitting a spike. Compared to the algorithm introduced in Jin et al., this TTS-STDP method is favourable in both memory and computational cost.

Most recently, Galluppi et al. (2015) have proposed a new concept for STDP calculations. In their method, neuronal and synaptic dynamic calculations are separated to different cores. This in theory separates the concerns and can allow better computational efficiency, at the cost of some latency in synaptic dynamics.

In all of these cases, it is claimed that the implementation of STDP is close enough to a canonical implementation (such as those in a software simulator) as to be negligibly different. In practice, this is true for longer run simulations. In the case of the NeuCube, however, there may be some small differences. It is noted in Galluppi et al. that a network's response to a stimulus may be slowed because it is not able to tune to the earliest spike groups of the input patterns. In developing NeuCube models for the SpiNNaker device we must be aware of this issue. Diehl and Cook (2014) claim that the current implementation of STDP is within 3% of benchmarks set in the software simulator Brian, which indicates that we should not be too concerned. Nevertheless, it is still valuable for us to be aware of the possible sources of variance in our experiments.

In practice it should make little significant difference; these 'slow starts' should theoretically be amortised out over the learning process of the network when presented with multiple patterns. However, it is worth noting that if we were to train two otherwise identical networks on a SpiNNaker and in a software simulator respectively, the synaptic weights evolved by the STDP processes may differ. Additionally, if we are using a Thorpe-inspired algorithm that weights early spikes more highly – such as the deSNN algorithm in Section 3.6.3.2 – this 'slow start' may affect which spikes are presented to the algorithm first and subsequently affect the evolution of the classifier in certain cases.

7.4.2.4 Random Number Generation

No hardware support for RNG is included in the SpiNNaker chip (Rast et al., 2010). Present estimates of random number generation times indicate that RNG in software on this device is somewhere on the order of $\approx$ 40x more computationally costly than a single neuronal dynamics calculation (M. Hopkins, personal communication, April 2015).

This indicates that the implementation of probabilistic neuronal or synaptic dynamics such as the probabilistic LIF model neuron discussed in Section 3.3.6 may not be feasible. Recall from that section that the pLIF requires three separate probilistic calculations: a probability that a spike will reach the postsynaptic neuron, a probability for that synapse to contribute to the postsynaptic neuron's membrane voltage, and a probability for that postsynaptic neuron to spike if its membrane threshold is reached. In the best case, a network of pLIF neurons would be $\approx$ 41x (RNG + neuronal dynamic calculation) more computationally costly than the same network of LIF neurons. This may be an unreasonable overhead depending on the use case of the network simulated.

The speed of RNG is increasing as the SpiNNaker group optimise the software, but unfortunately probabilistic neurons will always be second-class citizens in this architecture. It has been indicated that the next generation of SpiNNaker device (whenever this is officially proposed) will incorporate hardware RNG (S. Furber, personal communication, April 2015). In this case, the generation of a single random number would be performed in one clock cycle, indicating that pLIF-style stochastic neuronal models will be feasible.

7.4.2.5 Fixed Point Hardware

No hardware support for floating point arithmetic exists in the ARM968 processor cores used in the SpiNNaker device. Reportedly, this was a considered decision to reduce power consumption (Furber et al., 2013; Jin, Furber & Woods, 2008). A comprehensive discussion of the limitations of this approach is given in Hopkins and Furber (2015), where the authors introduce an efficient differential equation solver (required for neuronal dynamics calculation) for this hardware. As hardware arithmetic operations are only supported on fixed-point (integer) numbers, floating-point (decimal) operations must be implemented in software at considerable computational cost, or translated to a fixed-point representation.

The latter approach is used here. In practice, this is handled implicitly by the SpiNNaker's support software and is not generally a significant concern for our implementation of the NeuCube. See Section 7.4.2.1 for an example of a case in which a conversion to fixed-point caused an issue in the initialisation of the network. In the case that we wished to implement a custom neuronal model (*e.g.* the pLIF) or synapse model (*e.g.* the CNGM of Section 3.6.3.3), this would have to be a consideration. In particular, we would have to ensure that our representation function – *i.e.* the floating-point → fixed-point conversion process – provides sufficient resolution to represent the desired values, without introducing discontinuities. Inspiration should be drawn from the representation functions already implemented in the SpiNNaker support software as these are generally robust and have been created by the developers of the hardware itself.

7.4.2.6 Stochastic Spike Transmission

Spike transmission in the SpiNNaker device is a stochastic process (Rast et al., 2010). Recall that it is handled through multicast packets. These packets are source-routed; *i.e.* there is no handshaking or transmission/reception confirmation protocol. As a result, there is no quality of service guarantee. Spikes may be dropped in communication if the receiving chip or core router is saturated, and this packet loss can occur at any point in the transmission process. As a result, simulations run on the SpiNNaker cannot be considered to be deterministic; *i.e.* there is no guarantee that an identical simulation will return identical spike timing results. As spike transmission *in vivo* is naturally a stochastic process (Allen & Stevens, 1994; Hardingham & Larkman, 1998), and stochastic neuronal models have been shown to be more effective than deterministic models in certain contexts (Nuntalid et al., 2011), this is not necessarily a negative characteristic of the system. However, it is an important consideration, particularly when coupled with the other considerations introduced in this section.

In the case of the NeuCube, it is therefore theoretically possible that a reservoir simulation may diverge slightly from an identical network modelled in software, as this communication is not a deterministic process. However, it should be noted here that packet loss should only occur when the network links are saturated; a relatively rare occurrence under normal conditions. This link saturation is discussed further in the following section (Section 7.4.2.7).

7.4.2.7 Streaming IO with AER

An interesting feature of recent updates to the sPyNNaker system is the inclusion of live spike streaming. This feature allows us to stream spike timings from the host computer to the SpiNNaker device, and vice versa. If we intended to run the NeuCube in an embedded (*i.e.* on a robotic device) or continuous real-time manner (*e.g.* for BCI system control), it is possible for us to stream these spikes from our host computer to the remote simulation in an event-based manner. Live spike IO in sPyNNaker is interacted with in a syntactically similar manner to the `SpikeSourceArray` spike generator of standard PyNN. The functionality is provided in `spynnaker_external_devices_plugin.pyNN`.[5] Live spike IO is subject to the spike transmission constraints discussed above, as it utilises the same communication fabric and protocols.

At present, this feature has not been incorporated in the NeuCube as we have primarily been dealing with discrete simulations rather than continuous ones. Implementation of this feature would have to be considered in light of the discussion in Section 7.4.3.2, where we look at simulation repetition. A subsequent concern is that high speed spike streaming may saturate the communication fabric of the SpiNNaker device itself, leading to packet drops, and inconsistent simulation behaviour. For this reason, it is suggested that it is used judiciously, and until the higher speed SATA IO systems are developed further, avoided where possible. Practically, it would be trivial to implement; as few as three lines of code need to be changed. However, it must be considered in terms of its theoretical impacts first, and the context of the simulation second.

7.4.3 Modifications to the PyNN Implementation of the NeuCube

Recall from Section 7.4.1 that in the case of the SpiNNaker device, the interface used is actually a port of PyNN known as sPyNNaker. This sPyNNaker library is still relatively immature and incomplete when compared to the reference version of PyNN. In accommodating these limitations, a number of alterations to the original PyNN implementation of the NeuCube were made. Here we will discuss the most significant issues – the representation of network structure and distance, and simulation repetition – along with the rationale for making these changes. A number of

[5] For a practical demonstration see https://github.com/SpiNNakerManchester/PyNNExamples/blob/master/examples/external_device_examples/live_examples/spike_io.py

small changes were also made, which will be summarised in Section 7.4.3.3 The code listings in Appendix C incorporate these modifications.

7.4.3.1 3D Structures and Distance Representation in SpiNNaker

Recall from Section 6.4.4.3, and especially Figure 6.5, how distances should be modelled in simulations of SNN. In Section 6.4.3.1 how these have been implemented in PyNN is introduced, and why these have been implemented these manually rather than using the inbuilt PyNN methods is rationalised. In that section, we also allude to the fact that the maximum axonal delay is 16 ms.

This axonal length restriction is a limitation of the SpiNNaker system, which has a maximum spike ring buffer size of 16. For the principle of this buffer limitation and why it imposes a maximum axonal length on our networks, see the paragraph 'Simulating Axonal Delays with a Ring Buffer'. In addition, in that paragraph we will explain how these axonal delays are actually implemented in software.

For completeness, it is theoretically possible for a SpiNNaker device to support longer axonal delays through the use of what can be termed 'parrot' neurons. In short, a new neuron is generated external to the model. This new neuron n' has no internal dynamics and simply repeats ('parrots') its inputs. A neuron pair n_{pre} and n_{post} with a desired axonal delay of greater than 16 ms is connected via n' (*i.e.*, $n_{\text{pre}} \rightarrow n' \rightarrow n_{\text{post}}$), which is responsible for buffering the spikes from n_{pre} for the required time length, then sending them on to n_{post}. This results in a total maximum possible axonal delay of $(k-1)b$, where k is the number of neurons in the chain and b is their maximum buffer depth. The theoretical maximum axonal delay at present is 144 ms. While it is theoretically possible to implement longer axonal delays in this manner, this method is not yet robustly implemented and can cause some irregular behaviour in networks employing it. In addition, this requires additional processing cores to be assigned to handle these parrot neurons.

With this constraint in mind, axonal delays have been implemented to have a maximum equivalent length of 16 ms. Here, we will explain how this has been managed.

Distance Calculation Under Axonal Delay Constraints: In order for all networks to have a uniform maximum delay of 16 ms, we implement a simple normalising and scaling function.

Firstly, we calculate the inter-neuron axonal lengths of our network list $\mathbf{D}$, such that $d_{ij} \forall \in \mathbf{D}$ where d_{ij} is the Standardised Euclidean distance[6] between two neurons i and j, and $d_{ij} \in (0, 1)$. Secondly, we set the axonal delay t_{ij} proportional to d_{ij} with respect to the maximum axonal delay possible, *i.e.* that $t_{ij} \propto d_{ij}$, where $\max_t \in \mathbf{D} = 16$ ms. This can therefore be trivially calculated as $t_{ij} = d_{ij} \times 16$. This ensures that regardless of the actual maximum axonal length in a network, it will be scaled down to a delay range which can be implemented effectively in the SpiNNaker.

Of course, a caveat of this method is that inherently scales all networks to fit within a single proportional distance model, regardless of the actual distances. It is difficult to mitigate this issue in a generalised sense, as networks can have hugely variable connectomes. Depending on the network, axonal distances could be implemented without requiring this scaling, due to the fact that networks developed in the NeuCube exhibit high connectivity in a neuron's immediate neighbourhood, but few long distance connections. In graph theory sense, such networks exhibit small-world properties. If the maximum *actual* connection distance within a model could be found, we could therefore normalise relative to this distance, rather than the maximum *theoretically possible* connection distance. At present, this is not a significant enough issue to justify the extra development time, but as we find our models growing in scale, it may be worth exploring this issue in the future.

Functionally, these axonal delays must be simulated somehow in the system. which will be explained in the following section.

Simulating Axonal Delays with a Ring Buffer: Axonal delays (*i.e.* the time of spike propagation from one neuron to another) in the SpiNNaker device are generally considered to be negligible; a spike from an arbitrary presynaptic neuron should reach an arbitrary postsynaptic neuron in less than one machine timestep (≈ 1 ms). As mentioned in Section 7.4, spikes are transmitted across the SpiNNaker connection fabric through multicast packets containing a spike address and a time point. Due to the fact that actual propagation time of the 'spikes' is negligible in practice and is effectively constant (*i.e.* it is not directly proportional to the distance, as explained in Section 7.4.1), these spikes must be buffered somehow to simulate an axonal delay.

Here, the SpiNNaker implements a 'ring buffer' for each incoming synapse. In straighforward terms, a ring buffer $\mathbf{B}$ is a rotating circular buffer with a finite and fixed size. If there is incoming data, it is added to the current buffer location ($\mathbf{B}_{\text{first}}$). Simultaneously, the adjacent buffer location ($\mathbf{B}_{\text{last}}$) *prior* to the current location is

[6] Using the common formulation of Standardised Euclidean, $d = \sqrt{\sum (u_i - v_i)^2 / V[x_i]}$

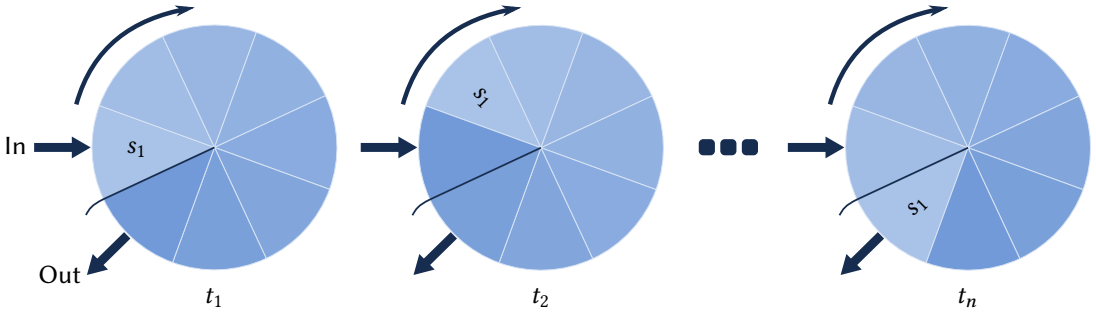


Figure 7.12: A visual intuition of how ring buffers are used in the SpiNNaker device to simulate axonal delays. A spike s_1 at time t_1 on an incoming synapse is inserted into the ring buffer. The buffer then rotates at each time step, eventually reaching time t_n (the simulated axonal delay), where the spike s_1 is popped out of the buffer and processed by the postsynaptic neuron.

inspected and any data contained within is removed. Regardless of whether data was injected or removed, the buffer then rotates, and this newly empty buffer location $\mathbf{B}_{\text{last}}$ becomes the new $\mathbf{B}_{\text{first}}$. See Figure 7.12 for a visual intuition of this process. Data can be injected in positions other than this $\mathbf{B}_{\text{first}}$, depending on its logical position in the buffer.

Alternatively, it can be conceptualised as a queue with a fixed size. At each time step, the bottom element in the queue is popped out regardless of whether it is null or contains a value. Simultaneously, a new element is pushed into the queue regardless of whether this new element is null or contains a value. The queue never changes length regardless of its contents, and the push/pop operations happen at each timestep.

A natural extension of this data structure, therefore, is the simulation of a fixed time-step axonal delay. Each synapse can represented by one ring buffer. The required buffer length can be found simply by multiplying the maximum time for spike propagation across the longest axon with the time resolution of the simulation system. In the case of the SpiNNaker device, this is typically 16 ms (16 ms distance × 1 ms time resolution), as discussed in Section 7.4.

At each time step, if a spike is received at the buffer controller, it is inserted into the buffer at its corresponding location. Simultaneously, the 'last' buffer location is searched for a spike, and if one is found, the spike is then processed by the neuronal dynamics calculation system. Regardless of whether a spike has been inserted or removed, the buffer then rotates at the next time step. This process repeats continuously. Depending on the length l of the buffer, our input spike has therefore

been delayed l timesteps by the time it is processed by the neuron, conceptually simulating axonal delay and the distance of the connection.

It may be possible that the length of the buffer (and thus, the memory footprint) could be adapted with the known length of the axonal connection. Shorter links could be represented by smaller buffers, as these do not require a long wait period to simulate the axonal delay. There does not appear to be any existing literature on adapative ring buffer size in simulations of SNN, and certainly none regarding the SpiNNaker device. Adaptation of the memory requirement for synaptic weights could offer a significant advantage when simulating large, highly connected networks, as these are typically bound by both the memory and processing cost of large numbers of synapses. These 'length-adaptive' ring buffers could significantly reduce the memory footprint of a short axon in the SpiNNaker system, which currently uses fixed-length ring buffers.

Taking inspiration from the SpiNNaker implementation, we have also briefly discussed the use of a ring buffer in the context of the NeuCube Core system. See (Section 6.6) for further details of this concept.

7.4.3.2 Simulation Repetition in SpiNNaker

As mentioned in Section 7.4.2.1, a challenge in the development of the NeuCube on SpiNNaker was the act of training a reservoir. The conceptual process for training such a reservoir in an SNN simulation system is to pass the first encoded sample to the network over a simulation run time, which is normally the same as the data sample length. After this data is passed, the internal state of the neurons is reset, any spikes currently in transit are deleted, and the network time reset to zero, while the synaptic structure (connections, weights) are retained. The next sample is presented, and this process repeats. This is logically a simple procedure, as shown in the pseudocode listing below.

```
intialise reservoir
for sample s with length t in data:
  input s to reservoir
  simulate for time t
  reset reservoir time, neuron potentials
```

Listing 7.3: Pseudocode implementation of an idealised case of reservoir training in software.

In the case of native PyNN, the `reset()` method is provided. This method takes care of the aforementioned steps, including resetting all of the neurons to their

resting state and the simulation time to zero. In that case, simulation repetition is straightforward. Simulation repetition in the original PyNN version of the NeuCube was implemented thusly.

Unfortunately in the case of sPyNNaker, no facility for the repetition of a single simulation is provided. To rectify this issue, we are left with implementing one of two options:

1. Concatenate the input data together into one spike train and perform one long simulation; or,
2. Manually reset the network, with the process detailed in Listing 7.4.

Concatenating the data together and passing one long spike train to the reservoir is one theoretical possibility. Viewed abstractly, this is how our brains learn. However, it is unlikely in practice to have a continuous train of stimuli of equal weight and properly ordered. Here, it is meant that natural input is unlikely to be presented in the correct order (*e.g.* 10 samples of the first class, then 10 samples of the second class), and that these presented samples will be of equal quality. When we view the reservoir, using this technique there will be overlap of signals due to the 'liquid' memory of the system. Spike trajectories from previous signals will still be propagating around the reservoir when a new signal is presented, and would cause an additive effect on the spike timings of the reservoir neurons. This feature, while useful in a real-world application, results in confusion of the recorded outgoing spike trains passed to the classifier, and in turn, reduced classification accuracy. It is worth exploring the significance of this effect in later experiments. For the moment however, input spike patterns will be separated and presented individually to the network. In this way, we can identify an initial benchmark for the system's accuracy, and analyse its behaviour independent of the possible confounding impacts of overlapping patterns.

Therefore, we must instead perform this network reset manually. Admittedly, this process is not difficult, but it does introduce additional, unnecessary delays to the training process. Instead of the process shown in Listing 7.3, we manually save the connectome after training, destroy the reservoir, reinitialise the reservoir with our saved connectome, and then pass the data. This process is shown below, in Listing 7.4, with the overall process described in Listing 7.1.

```
for sample s with length t in data:
  intialise reservoir
  if saved connectome:
    load connectome
  input s to reservoir
```

```
simulate for time t
save connectome
destroy reservoir
```

Listing 7.4: Pseudocode implementation of reservoir training under the limitations of the the SpiNNaker device.

As discussed in Section 7.4.2.2, IO for these devices is currently constrained to 100 Mbps Ethernet. In the case of the connectome, this is stored in the large SDRAM on each chip. Recall that the SpiNNaker chips communicate through multicast packets. In order for the connectome to be extracted from the device, the DMA controllers for the management chip must make access to this SDRAM, package it, pass it to the chip router, and in turn to the system bus and out through the Ethernet port. Each step in this process induces additional delays. Observations from the work in this thesis indicate that at present, the retrieval process for a small simulation ($\approx$ 1500 neurons each with a mean of 10 synapses) takes around three seconds in total, from request to completion. A similar amount of time is taken when loading the connectome onto the device. The actual simulation runtime is dependent on the parameter passed to the `run()` method of PyNN (in this case, sPyNNaker). See Section 7.5 for a more comprehensive discussion of the time cost of simulations.

Of course, it is theoretically possible to reduce these load times by creating simpler networks. However, the example given above is extremely simplistic with respect to SNN, particularly when we consider the number of synapses. Typical models have between 100–10,000 synapses per neuron, with the human brain having a mean of around 1,000 (Kandel & Schwartz, 2000). This issue would be more pronounced with larger networks, as SNN follow a power law with respect to the branching factor of a single neuron. Clearly as we are intending to use the SpiNNaker to provide the capacity for larger NeuCube reservoirs, this issue will quickly become untenable.

In discussion with the developers of SpiNNaker, it was made clear that this issue was not considered in early development of the system. This is due to the fact that it was originally intended as a platform for computational neuroscience, and not neuromorphic applications. In the case of computational neuroscience applications, such a device would be set up and a long term simulation run, with the synaptic weights left to evolve online and interacted with only at the end of a single long simulation (if at all). Here, they would be more concerned with spiking behaviour of different cortical areas – data which is easily streamed out online – rather than weight evolution. Online spike streaming is discussed in Section 7.4.2.7.

The ability to extract the synaptic weight matrix from the hardware was initially missing from the system. Until this was implemented, the implementation of NeuCube on SpiNNaker was not feasible, as there was no facility to implement the learning processes detailed in Section 4.2. The SpiNNaker group implemented this feature after the April 2014 SpiNNaker Workshop. This issue directly relates to the challenges discussed in Section 7.4.2, particularly the sections on development ecosystem maturity and IO limitations.

Therefore, two significant performance issues are raised by this methodology:

1. Time cost of generation of the simulation on the host computer, and
2. IO speed over Ethernet.

The IO issue is a difficult challenge to overcome. As mentioned previously, the intention of the developers is to make the SATA links on each SpiNN-5 board available as general purpose IO devices. Utilising the SATA links will raise the maximum bandwidth from 100 Mbps to a theoretical maximum of 3.1 Gbps. This issue is discussed in more detail in Section 7.4.2.2. Additionally, the simulation generation process (partitioning & placing, *etc.*) induce extra delays. Simulation generation delays are unnecessary as in this case, the network placement will always be identical; the same neurons will be partitioned to the same cores.

The simulation repetition set up time issue can be avoided with some small changes to SpiNNaker. At the end of a simulation, the connectome data is retained on the SDRAM; a cleanup process run during the sPyNNaker `end()` method (called at the end of a simulation) invalidates this data when it is no longer needed. However, it is theoretically a straightforward matter to simply reset the neuron states and internal clock timer to their initial states, and retain the connectome in memory. This would be the optimal resolution to this issue, as then the IO constraints for saving and loading a network structure are amortised over a whole simulation rather than each sample. Moreover, the simulation generation process will not need to be repeated, as the neurons will not require partitioning & placing again. They simply remain in their intial memory locations. Essentially, we then only need to implement lines 5–9 of Listing 7.1. Unfortunately, the scale of this change necessitates implementation by those intimately familiar with the sPyNNaker software, which puts it outside the scope of this thesis. The addition of this feature is currently under development by the SpiNNaker group[7] but at the time of writing this thesis it was not part of sPyNNaker

[7] https://github.com/SpiNNakerManchester/sPyNNaker/tree/work_flow_multi_run

stable. This feature should be considered stable and added to the main sPyNNaker code near the end of 2015 (A. Rowley, personal communication, December 2015).

As an alternative, the SpiNNaker group have been experimenting with the ability for the connection structure to be generated on-board the device (A. Rast, personal communication, April 2015). In effect, instead of this being generated on the host computer and then being transferred to the SpiNNaker device, an abstract representation of the network would be transferred to the device, and the SpiNNaker itself left to generate and place the network and its connections. This should improve network setup speed, by reducing the effect of the external IO bottleneck. No practical implementation of this feature has yet been demonstrated, but it may be of use to us in the future. For the moment though, we must make do with the mitigation strategies discussed above.

7.4.3.3 Summary of Important Minor Changes

Summarised here are the important minor alterations to the PyNN implementation of the NeuCube in order to run it on the SpiNNaker.

Import Statement: To run our model on the SpiNNaker we can `import sPyNNaker` directly and remove the `import pyNN.nest`, or `import pyNN.spiNNaker` to keep visual compatability instead. The latter has been used here, as it is then possible to poll the system for the installed simulators and using Python's `exec` function, dynamically set the system to be imported. This is visually more understandable when changing the package imported, rather than the whole library.

Core Limits: As both the neuron dynamics and STDP are calculated on the same device, the addition of plasticity can cause the SpiNNaker cores to be overloaded in certain circumstances. In the case that we have extra cores, we can suppress this small issue and ensure that the behaviour of the board is more consistent by manually defining the maximum number of neurons partitioned to each core. We implement this in sPyNNaker with `set_number_of_neurons_per_core("IF_curr_exp", 128)`.

7.4.4 Operation of the NeuCube on SpiNNaker

The actual operation of the NeuCube code on the SpiNNaker is identical to that of the standard PyNN code given in Section 6.4.2. Initially we write the configuration file with the desired network structure and settings, here denoted as `config_file.json`

in the `model_data` directory. An example of such a configuration file in the JSON format is given in Appendix F. We also need to make one minor code change, as described in Section 7.4.3.3, where we change the import statement. In the future, this should be an automated step where the desired simulator is drawn from the configuration file.

From here, the internal steps are the same between the standard PyNN implementation from Section 6.4 as this version. Subsequently, we call the main method of the system giving the configuration file as the argument, *i.e.* in the root directory of the project, call `python main.py model_data/config_file.json`. From there, the process is as described in Section 6.4, with the obvious caveat that the actual simulation will take place on the SpiNNaker device rather than the host computer.

In fact, this is largely the reason the NeuCube has been implemented in PyNN. The code created using the PyNN library is portable, and can be moved between computation platforms with little to no code changes, and ensures relatively constant simulation dynamics. In this way, the discussions in Chapter 6 (and particularly, Section 6.4) apply to both the software and neuromorphic hardware implementations of the system.

7.5 Empirical Comparison of SNN Simulation in Software and Hardware

At this point, it is valuable to show some comparison between key dynamics in software and neuromorphic hardware simulation of the NeuCube. In this section, we discuss some performance profiling and behaviour of these implementations.

The same execution time profiling technique used in the identification of JIT compilation's performance benefits (*cf.* Section 6.4.3.2) is used here, with the addition of some memory use metrics. To profile the memory dynamics, the `memory_profiler` library[8] is used, which queries the OS kernel to determine memory requests at each method call or line operation.

The actual operation profiled is the generation and single simulation of a number of otherwise identical networks which vary in scale. These networks are a series of arbitrarily shaped (cuboid) networks at a roughly linear scale with regard to the network 'sides' were generated. These networks were then run three times each in the available versions of the NeuCube, on both the Brian and SpiNNaker simulation

[8] https://github.com/fabianp/memory_profiler

platforms. Each network has 14 inputs structured based on the Emotiv EEG device electrode locations. Here, to simplify matters, these inputs are not real data, but Poisson trains. In this case, we would consider the step in question to be the initial 'training' of a network on a single sample. Data transfer time is incorporated into these metrics, although it is also possible to stream it on-line in the SpiNNaker device.

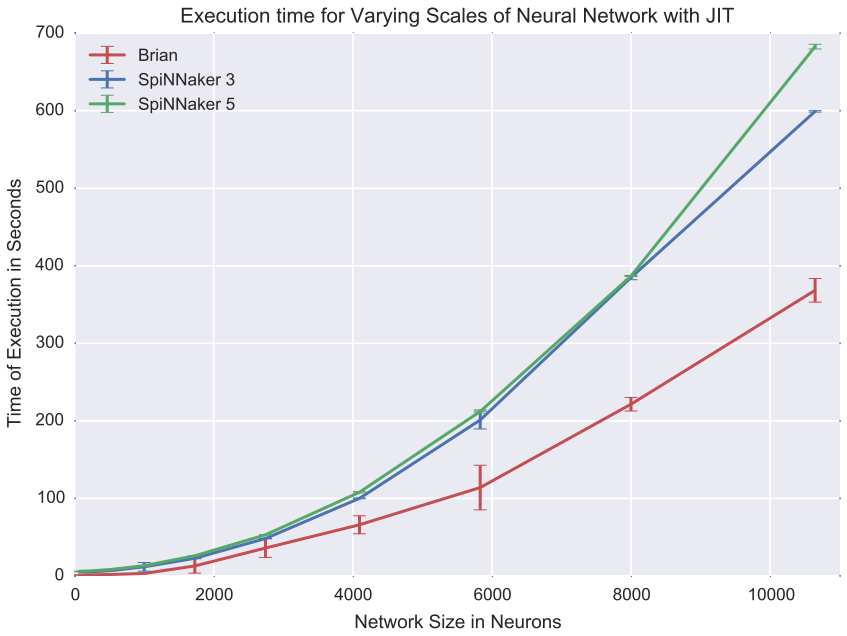
Some interesting dynamics are revealed through this profiling. See Figure 7.12 for execution time dynamics, and Figure 7.13 for examples of memory consumption dynamics. In the following sections, some specific features of these dynamics are discussed, including the apparent (and somewhat counter-intuitive) illusion that software simulation is actually more efficient than neuromorphic hardware simulation.

The results in this section should be treated as illustrative rather than definitive; there are simply too many variables to account for in experiments of this type. Different network shapes, neuron parameter selections, connection schemes, and so on will affect the dynamics of the network simulation, particularly as the network complexity and scale increases.

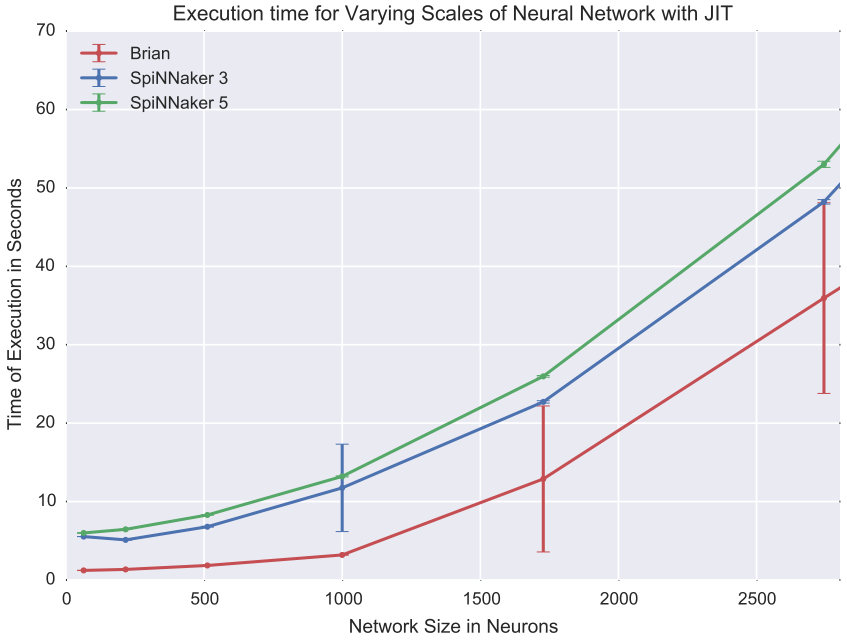
7.5.1 Execution Speed Dynamics

Perhaps the most obvious conclusion to draw from this first set of plots in Figure 7.12, is that the premise of this section – *i.e.*, that neuromorphic hardware implementations are more efficient than software implementations of the same network – is incorrect. While this may in fact be true under certain conditions, the traces in this figure do not tell the whole story. Most significantly, the apparent lower performance of the SpiNNaker implementations is an artifact of the long IO times presently encountered on that platform and discussed here in Section 7.4.2.2 and Section 7.4.3.2.

As mentioned previously, the IO available to the SpiNNaker devices is presently based on a 100 Mbps Ethernet connection. For this device's original intended purpose of long computational neuroscience simulations, this limitation is acceptable at present, as model loading is a very minor time consideration compared to the length of the actual simulation. However, in our case, as we are required to load and download models frequently (*cf.* Section 7.4.3.2), the very significant performance increases in *simulation time* are being masked by this *IO time*. With the assumption that these IO issues are resolved, the execution time plot should look more like Figure 7.12c, where the execution of a simulation on the SpiNNaker device is more-or-less constant with some minor IO variablity based on the network scale, and of course – proportional to the simulation run time selected. In fact, some evidence of this variability can

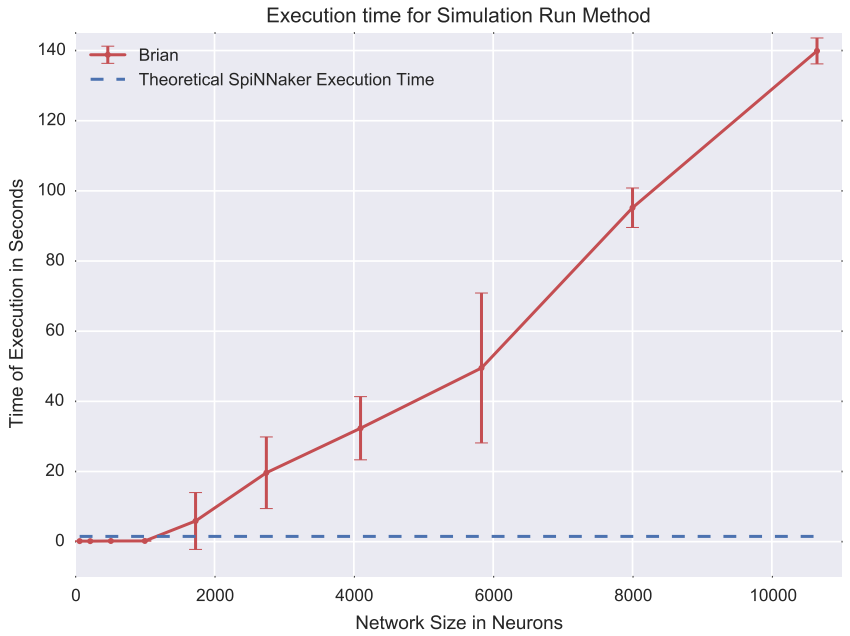


(A) Execution time for the NeuCube reservoir on different computational platforms.



(B) Detail of execution time for the NeuCube reservoir on different computational platforms.

be seen in the error bars of these figures. Otherwise identical simulations run in Brian are much more variable in their run times. This is likely to be an artifact of the host device. Since Brian runs in Python system running locally on the host device, a number of issues may have affected the variability of these results: thermal throttling on the device, garbage collection overheads in the Python environment, 'warmup'



(c) Execution time for the actual simulation step of the NeuCube reservoir.

Figure 7.12: Overall execution time for a single simulation of the NeuCube reservoir on a software simulation and the two available SpiNNaker devices (Figures 7.13a and 7.13b). Here the general indication is that software simulation is faster for these networks. The actual simulation time minus the current IO constraint is shown in 7.12c. Here, the execution time for a network is constant regardless of the network scale. This phenomemon is explained in text.

of the just-in-time compiler, and so on. It is a theoretical benefit of the hardware approach to SNN simulation that run times and performance are more predictable, as they are not subject to the vagaries of a general-purpose computing device like a laptop.

In terms of other neuromorphic systems such as the cxQuad discussed in Section 7.3, we would see much the same computational time; these, too, operate in biological time (*i.e.* there is a 1:1 mapping of simulation time to real-world time), which is independent of the scale or complexity of the network it simulates. Again, then, we would likely see that the software simulation is likely faster for very (trivially) small networks, while for larger or more complex networks these systems would likely be faster. It is reasonable to assume that the total time required for the same simulation will be lower for the cxQuad than for SpiNNaker, as it is a simpler interface and the network upload process is more straightforward. This is of course, with the obvious caveat that the SpiNNaker can simulate a larger number of neurons at present, and is more easily configured.

This is, of course, an idealised case; however, the operation of the SpiNNaker ensures constant time execution of networks, with no slowdown based on the network scale. The actual operation of the simulation in all of the cases presented in Figure 7.12 on the SpiNNaker device was on the order of 1.5 seconds (*i.e.*, that shown in Figure 7.12c). By comparison, the trace shown there for the Brian simulation is the actual time for the execution of the simulation, as there is no issue of IO. In this way, as the network size increases, the execution time for software simulation will increase proportionally until it is untenable, while for neuromorphic hardware it will stay constant (with considerations for IO).

Presently, we can represent the observed values by a quadratic function, as:

$$6.31627 \times 10^{-6}x^2 + 0.00050115x + 6.19627 \tag{7.4}$$

which has an agreement of $R^2 = 0.999986$. As a result, if we are to look towards truly large scale networks such as the 2,500,000 neuron set representing 1 mm^3 of brain volume per neuron in the MNI data set, the runtime soon becomes daunting. This particular case indicates a runtime of around 3,947,681,284 seconds, or around 125 calendar years on the particular host computer. However, as mentioned, this is a property of the $O(n^2)$ complexity for the network generation step discussed in Section 6.4.3.1. Additionally, this is in the single-threaded case. It is relatively trivial in a theoretical sense to optimise this network generation step, through the use of vector operations and multi-core (or possibly GPU) processing, as this is a highly-parallel task. At the point we begin to look at networks of this size and complexity, we should consider optimisation of the network generation code here.

It is also valuable to emphasise that a neuromorphic hardware solution is not ideal for all networks. Indeed, in the case of small scale networks, it is faster to implement these in software simulation. This principle is illustrated in Figure 7.12c, where the software simulation is faster for networks of under around 1,200 neurons for this particular network.

An interesting dynamic of the SpiNNaker devices is exposed by Figure 7.12. For smaller networks – *i.e.*, those small enough that they can be simulated on the SpiNN-3, around 16,000 neurons – it is faster to simulate on the smaller 4-node SpiNN-3 board than the larger 48-node SpiNN-5 board. This may be a property of the less complex board control mechanisms of the smaller device; model IO for the SpiNN-5 is mediated by a system of FPGA. This control system is necessary to manage the complexity of simulation of larger networks, but may induce additional delays for

smaller networks. This indicates that for smaller networks, we should first consider a software platform, followed by the use of the smallest neuromorphic device which supports the network scale we wish to simluate. This of course, will also serve to encourage lower power consumption where possible, as the smaller devices typically consume less power.

This idealised case is not yet the reality. IO speeds on the SpiNNaker device are presently improving, in line with the discussion in Section 7.4.2.2. Once this issue is resolved, the constant-time simulation advantages of this platform will be more apparent for large networks.

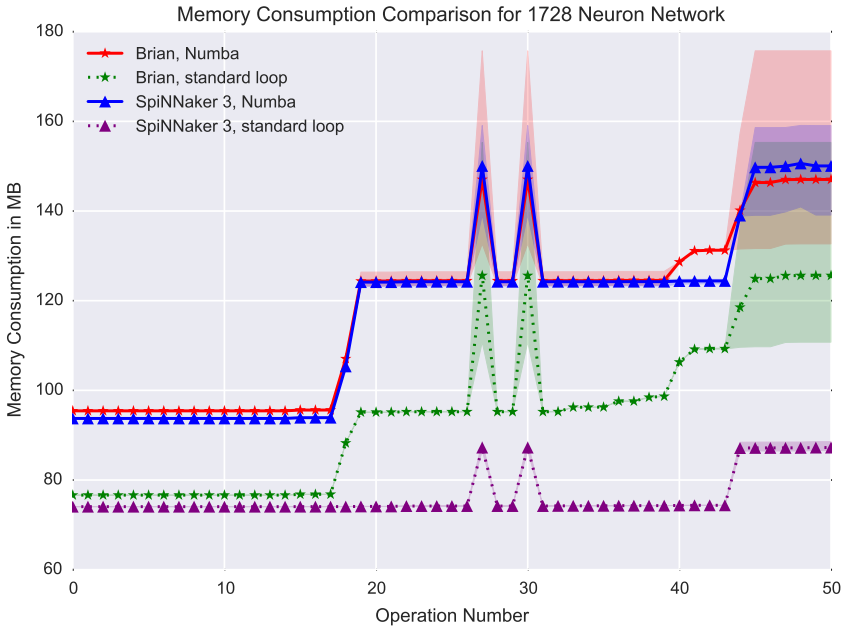
Additionally, we should note here that the IO issue on the SpiNNaker will soon be a one-time cost per simulation; with the added ability to reset the network dynamics without reloading the network descriptor onto the device, subsequent samples or input patterns can be presented to the device online, with constant-time processing as in Figure 7.12c. This feature is discussed in Section 7.4.3.2, where it was mentioned that an upstream change to the sPyNNaker code will make this reinitialisation process considerably faster. In this case, we will still have some time-cost in intialising the network, but more or less constant time execution of the actual repeated simulations. Contrast this with the time dynamics of a software simulation, which will grow proportionally with the network complexity. In this way, a network simulated on the SpiNNaker device will still be faster than an equivalent software simulation, even with the present IO limitations. This is a significant advantage of the neuromorphic approach, and one which motivates the work in this thesis.

7.5.2 Memory Use Dynamics

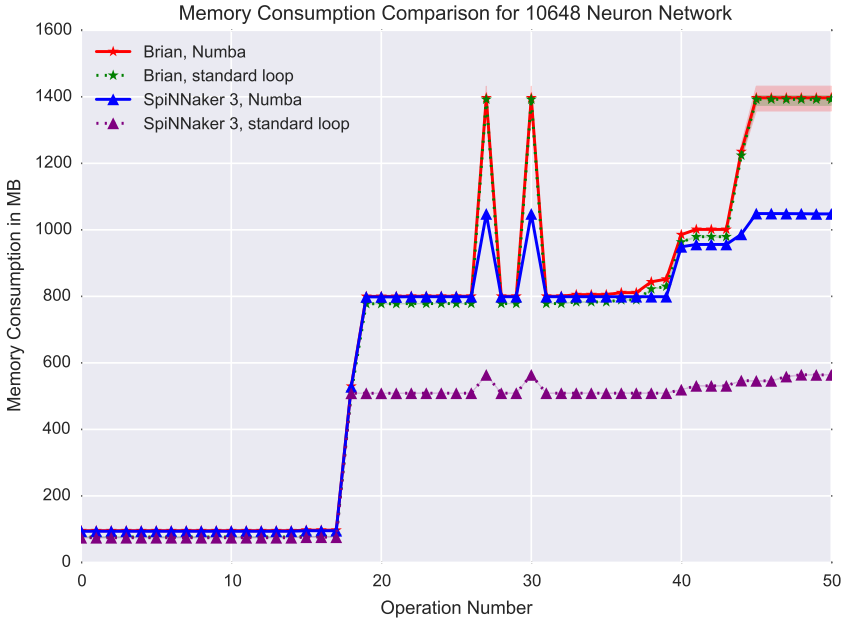
A subsequent concern to the execution time of the system, is the memory consumed by the system on the host computer. While this is generally not a significant concern, due to the branching factor exhibited by SNN at large scales it may become challenging to manage this demand. To quantify the extent of this issue and identify whether it is a concern for us, the experiment of Section 6.4.3.2 has been repeated with memory consumption as a focus, rather than computational time. Here, we show the 'operation' number as the time factor. This operation number generally corresponds to the operation or line number of the `NeuCubeReservoir.py` file.

An example of the memory consumption history generated by the `memory_profiler` tool is too long to include here, but a representative example is available online.[9]

[9] https://gist.github.com/nmscott/164ab24ee110fce856ad



(A) Memory consumption for a 1728 neuron NeuCube reservoir on different computational platforms.



(B) Memory consumption for a 10648 neuron NeuCube reservoir on different computational platforms.

Figure 7.13: Memory consumption on the host computer over the progression of the application. Operation numbers represent method calls or line operations in the NeuCubeReservoir.py file. Memory consumption is generally lower overall when using a neuromorphic computing platform. Shading represents the 95% confidence interval for that value.

In Figure 7.13 two representative plots of the memory consumption of a NeuCube reservoir per operation are presented. A more comprehensive list of memory consumption dynamics graphs is given in Appendix E. It is important to note here the stochastic nature of these measurements. As they are queried from the kernel, they are subject to some varability based on background processes and the vagaries of memory management induced by the Python interpreter. In this case, a 95% confidence interval indicated by shading has been included in these plots. Additionally, these results are illustrative for these particular networks simulated; a network with a larger or smaller number of synapses would occupy a proportionally larger or smaller amount of memory.

As a general rule, memory consumption is lower when using a neuromorphic hardware platform for network simulation, as compared to software simulation. This can be reasonably expected given the lower computational demand on the host computer in this case, as processing and memory requirements are offloaded to the hardware device instead of being solely addressed by the host device. The actual total memory consumption (across both the host device and hardware simulator) tracks with the scale of the networks and will vary depending on the simulation device used; in general, large networks require more memory, and smaller networks require less. Again, this is not an unexpected dynamic, as the networks need to be stored *somewhere* when they are generated. Memory variablility across runs is reduced, although this may be an artifact of the fact that the background fluctuations are proportionally less significant to larger networks.

The first significant rise in memory consumption is encountered when generating the network structure (neuron locations and connectome). This baseline would fluctuate with the size of the networks and the complexity of its connections. Subsequent peaks are seen when entering the control loops, which preallocate space for the files needed when performing multiple runs of the same network (*i.e.*, learning). These are then freed when the interpreter predicts that they will not be used. The next significant rise is seen when actually running the simulation; in this case, we would naturally expect to see a more significant rise in the memory consumption of Brian. However, this is not the case, as the setup process has already prepared the requisite memory for these steps. Similarly, we see no significant rise in the memory consumption of the SpiNNaker version, as at this point the model is copied to the device. The final significant increase is exhibited when reading back the synaptic matrix and spike times.

Interestingly, memory consumption is generally lower when employing standard Python loops, rather than the optimised form provided with the JIT compiled loops. This is likely due to the `numba` library trading processing time for a larger memory footprint, although it is not possible to say for sure. We should make the decision here whether this increased memory consumption is worth the improved calculation times afforded by this system; in almost every case it would be appropriate, as processing time is the key metric here.

In general, memory consumption for most networks will not be a concern. In the case that we are dealing with large (15,000+) neuron networks, we would likely be performing these calculations on a relatively high performance host computer, and it is unlikely that we would encounter a situation where the memory consumption was a halting issue. In that case, a more optimised solution for the calculation of the network structure would be appropriate.

7.6 Ideal Hardware Characteristics for Neuromorphic Implementations of the NeuCube

From the discussions in this chapter, it is possible to identify some idealised characteristics for a neuromorphic hardware platform on which to implement the NeuCube. Here, we introduce a short discussion of the key features desired for the NeuCube specifically, rather than a generalised neuromorphic platform.

1. Digital platform implementation

The choice between digital and analog (or alternative implementations such as FPGA) is a highly contextual one. However, given the general features of the NeuCube and its inital areas of application, it is apparent that a digital neuromorphic platform such as the SpiNNaker is a more appropriate choice at present. This is primarily due to the ease and flexibility of network changes on such a platform; as noted, this type of implementation emphasises flexibility over power efficiency. This is appropriate in the inital stages of developing a NeuCube model, which can then in theory be ported to an analog or alternative platform once the bulk of the network exploration has been performed. Additionally, a digital platform currently has some advantages in terms of neural and synaptic model selection and volume.

2. High maximum number of synapses per neuron

A relatively high possible number of synapses per neuron is advantageous at present, as the behaviour of the NeuCube when limited to low numbers of synapses has not

yet been established or analysed in any depth. NeuCube models established in the literature presently have a higher number of synapses per neuron than is possible on devices such as the cxQuad or Zhejiang FPGA. This is not to say that the NeuCube cannot operate under this constraint; merely that its behaviour under this constraint is not yet studied.

3. Macroscopic neuron models supported, particularly the LIF neuron

All exploration of the NeuCube has at present been performed utilising the LIF model neuron. It is therefore logical that we should expect a simulation platform to support this model type, until such time as the optimal neuronal model has been established for a given use case in the literature. Here, we can make use of most of the major neuromorphic hardware devices, with some exceptions. Additionally, it is preferable that this neuron type is interchangeable, in order to establish these optimal neuronal model selections.

4. Faster-than-clock-time model evaluation

Ideally, models should be able to be simulated and evaluated in faster than real-time for simulation purposes. Real-time operation is beneficial when we are addressing real-world stimuli, but is not optimal for the initial simulation, training, and evaluation of NeuCube models. At present, during the process of establishing the NeuCube's properties, faster than real-time simulation is optimal. The option to transition the model to a real-time device or device mode is beneficial should implementation in a real-world scenario be required.

5. Configurable synaptic delays or other method of axonal distance representation

The importance of configurable synaptic delays have been addressed in Section 6.4.4.3, where it was established that without some method of managing axonal delays, 3-dimensional networks could not effectively be represented in a simulation. Therefore, some mechanism for representing distance should be implemented. In the case of the SpiNNaker and similar devices, this is typically implemented as some form of input buffer between neurons.

6. Preferably, the ability to run multiple simulations at once on the same device

We effectively compose a NeuCube model of two or three primary neural simulations: *a)* Optionally, the encoding of input data into spike trains, which may take the form of a small neural network in the case of population encoding; *b)* The simulation of the 3-dimensional reservoir network; and *c)* The output classifier, which will typically be

SNN based. Here, we typically need to simulate these systems separately. This may mean that we simulate them sequentially (*i.e.* encoding, reservoir, output), but it is also entirely correct to simulate them simultaneously; they are, after all, operating on the same data in the same time scale. This is advantageous when addressing real-world stimuli, where as near to instantaneous responses are required. As a result, it would be useful if the neuromorphic device we select has the capacity to run these multiple simulations simultaneously. Here, it would be especially useful if these simulations could communicate with each other asynchronously, without the need for a mediating host device. As a secondary benefit, this would potentially allow us to run parameter space searches or network ensembles in a more efficient way.

Power consumption has been omitted from this list. Despite this being a major motivating factor in implementing the NeuCube on a neuromorphic hardware device, it is not at present a significant concern. More emphasis should presently be applied to the identification of system dynamics and optimal network design, before improvement in power consumption is considered in depth.

Obviously, this is a non-exhaustive list of the features considered beneficial for neuromorphic simulation of the NeuCube. However, this should go some way towards providing a framework to make neuromorphic device selections in the future. If we are to apply these considerations to the presently available neuromorphic hardware devices, the assessment is that the SpiNNaker device is the most appropriate currently, primarily due to its flexibility in representing large-scale neural network models. This, however, is subject to change as the behaviour of the NeuCube is better understood. It is entirely feasible that in short order, neuromorphic VLSI systems such as the cxQuad will be advantageous due to their low power operation.

7.7 Chapter Summary and Conclusion

In this chapter, we have presented a treatement of the current state-of-the-art in neuromorphic hardware systems, which are dedicated hardware devices for the simulation of SNN. These devices have a number of benefits, most particularly low power consumption and asynchronous real-time processing. They are also generally limited in the number and type of neuron and synapse models which can be implemented, which imposes some constraints on development of applications for them. Particular concerns are the small resolution of synapse weights and the limited number of synapses generally available on such systems.

A preliminary implementation of the NeuCube on an FPGA-based neuromorphic hardware device is introduced. This system presently has a number of constraints including the fact that it contains no facility for on-line learning or STDP, which must be rectified before the NeuCube can be effectively implemented here.

A more theoretical discussion of a NeuCube implementation on a subthreshold analog-digital hybrid neuromorphic ASIC was also introduced. This device, the cxQuad, provides up to 9,000 neurons at extremely low power. At present, the support software is not complete enough to justify the development time it would take to develop a concrete version of the NeuCube. However, once it is complete, it is our intention to explore this computational platform thoroughly. This system offers a number of new applications for the NeuCube, including in an embedded context, due to its low power.

A complete version of the NeuCube for the SpiNNaker device has been introduced here. A review of this system and its advantages for the NeuCube is given. This implementation is written in the sPyNNaker port of the PyNN library, which is unfortunately still incomplete. A number of considerations, including physical limitations like the speed of the Ethernet-based IO of this system and its fixed point hardware, are discussed here, and contextualised in how they affect the behaviour of the NeuCube.

This chapter is one of the most significant in this thesis, particularly as it pertains to the future of the NeuCube. It is completely infeasible for us to look to more complex datasets without a functional and comprehensive implementation of the NeuCube for neuromorphic systems. These systems provide the capacity for almost arbitrary scaling of NeuCube networks, at much lower power consumption than a software simulation on standard commodity computing hardware. The considerations in this chapter for these devices are also generalisable; they are not specific to the NeuCube, but can be used for any large scale SNN architecture which seeks to exploit neuromorphic hardware.

Contributions of this Chapter

1. A review of contemporary and emerging neuromorphic hardware systems.
2. The introduction of an implementation of the NeuCube architecture for generic neuromorphic hardware.
3. A preliminary implementation of the NeuCube for the Zhejiang FPGA-based neuromorphic hardware system.

4. A preliminary treatment of the considerations for implementing the NeuCube on the cxQuad and ROLLS neuromorphic VLSI ASIC hardware systems.
5. A treatement of the considerations for implementing the NeuCube on the SpiNNaker neuromorphic hardware device, including:
   (a) Theoretical constraints and advantages
   (b) Code and ecosystem maturity challenges
6. Empirical evidence of the behaviours of both neuromorphic and software simulations of the NeuCube.
7. A concrete implementation of the NeuCube written in the sPyNNaker port of PyNN for the SpiNNaker neuromorphic hardware device.
8. Identification of an open area of research in minimising memory footprints for axonal delay simulation using length-adaptive ring buffers.

Peer Reviewed Literature With Direct Contributions from this Chapter

1. Kasabov, N., **Scott, N. M.**, Tu, E., Marks, S., Sengupta, N., Capecci, E., Othman, M., Doborjeh, M., Murli, N., Hartono, R., Espinosa-Ramos, J.I., Zhou, L., Alvi, F., Wang, G., Taylor, D., Feigin, V., Gulyaev, S., Mahmoud, M., Hou, Z.-G. and Yang, J. (2016). Evolving Spatio-Temporal Data Machines Based on the NeuCube Neuromorphic Framework: Design Methodology and Selected Applications. *Neural Networks*. Special Issue on Learning in Big Data. Elsevier. doi:10.1016/j.neunet.2015.09.011
2. **Scott, N. M.**, and Kasabov, N. (2015). Feasibility of Implementing NeuCube on the SpiNNaker Neuromorphic Hardware Device. In *13th International Conference on Neuro-Computing and Evolving Intelligence*. February 19–20, Auckland, New Zealand.
3. **Scott, N. M.**, Kasabov, N., and Indiveri, G. (2013). NeuCube Neuromorphic Framework for Spatio-Temporal Brain Data and Its Python Implementation. In *Proceedings of the 20th International Conference on Neural Information Processing*, 3–7 November 2013, Daegu, Korea. Springer. doi:10.1007/978-3-642-42051-1_11

Chapter 8

Conclusions

> Emergent behavior is that which cannot be predicted through analysis at any level simpler than that of the system as a whole... It, by definition, is what's left after everything else has been explained.
>
> — George B. Dyson
> *(Darwin Among the Machines: The Evolution of Global Intelligence*, 1997)

This thesis has addressed a number of different areas of both theory and practice in SNN, and advanced the current state-of-the-art in a number of ways. In this final chapter, we firstly discuss the primary contributions it has made to the literature. Subsequently, we address the research questions posed in Section 1.1, and how these have been resolved in the thesis. The key caveats and limitations of this work are then examined, along with a review of the future works and open questions identified. Finally, this thesis closes with some thoughts on the nature of this study and its overall contributions.

8.1 Novel Contributions

At the end of each chapter in this thesis, the specific contributions of that chapter have been enumerated. Similarly, a list of peer-reviewed publications containing contributions from that chapter have been summarised in the same way. Therefore, here we briefly summarise the key contributions, those which have made the greatest contribution to the state-of-the-art. The interested reader is directed to those summary sections for a more comprehensive list. The primary novel contributions of this work follow.

Primary Contributions

1. A general design methodology for NeuCube based SNN systems.

By this, we mean the work of Chapter 5, wherein a methodology for the design of a NeuCube based SNN system is developed. A step-by-step methodology for the decision making process behind the generation of encoding schemes, structured reservoirs, and output devices, has been provided in a generalised sense.

As the development of NeuCube systems – and indeed, SNN systems in general – is a largely heuristic process, it is desirable to impose as much rigor as possible. This methodology should go some way towards providing a systematic method of designing NeuCube based SNN (and indeed, any reservoir SNN) in a meaningful, informed, and reproducable way.

2. A general implementation framework for the NeuCube.

A software design and implementation framework is one of the most significant contributions in this thesis (*cf.* Chapter 6). This contribution should ensure that future development of the NeuCube framework is implemented in a modular, extensible, and most importantly – theoretically accurate fashion.

Here we have provided the requisite software architecture for an object-oriented version of the NeuCube, including class diagrams and a list of necessary methods. Additionally, development guidelines following the Unix design philosophy have been introduced.

3. A generalised NeuCube realised in Python using the PyNN simulator interface library.

A concrete implementation of the framework established in the previous contribution has been developed. This concrete implementation (*cf.* Section 6.4) is developed for cross-platform use in manner which is modular and accurately reflects the theory inspiring it. This cross-platform computational ability will be extremely useful in future development of the NeuCube, as it allows the system to be applied in a number of different contexts (commodity PC or laptop, cluster, supercomputer, *etc.*) without significant changes to the code, and with the knowledge that the system behaviour is predictable and traceable .

In the future, this software implementation should serve as a template for future development of NeuCube systems. A number of interesting challenges were addressed

in this development, including the adaptive mapping of network structure and input locations.

4. Preliminary exploration of a NeuCube realisation for the Zhejiang FPGA neuromorphic hardware system.

The preliminary steps have now been taken to implement the NeuCube on a dedicated FPGA-based based neuromorphic hardware system (*cf.* Section 7.2). This implementation will allow the NeuCube to be applied in relatively low power environments.

Additionally, it is the first meaningful model developed for the Zhejiang FPGA, which is a system still very much in its infancy. While a complete application has not yet been demonstrated on this computational platform, the preliminary work has been completed. With some further development of the FPGA itself to address the issues raised in Section 7.2, this collaboration promises to be a significant contribution to the state-of-the-art.

5. Preliminary exploration of a NeuCube realisation for the cxQuad and ROLLS VLSI ASIC neuromorphic hardware devices.

Another contribution in the field of neuromorphic hardware systems – and, arguably a more significant one than the FPGA system discussed above – is the preliminary work for an implementation on the INI neuromorphic ASIC devices. A discussion of the considerations for a NeuCube implementation on such a device is given in Section 7.3. In short order, the system support software of these devices should reach a sufficient maturity to be released to external (non-expert) users.

An implementation of the NeuCube on this particular computational platform would be revolutionary in the types of applications which will then be available to us. The extremely low power, real-time computation of such neuromorphic hardware is an ideal fit for robotics and BCI applications. The NeuCube is shown to be effective in a neuroinformatics context; in conjunction with a computational platform like the cxQuad, it is ideally applicable for robotics control in rehabilitation and assistive technologies.

6. A NeuCube realisation for the SpiNNaker neuromorphic hardware device.

One of the largest contributions of this thesis is implementation of a NeuCube architecture for the SpiNNaker neuromorphic hardware device; one of the first such meaningful applications of this computational platform. Here (Section 7.4), we have

discussed the considerations for developing a NeuCube on this platform, particularly with regard to the constraints of the hardware and modifications to the existing systems.

This implementation is advantageous as it allows for the NeuCube framework to scale to arbitrarily large network sizes. This is a necessary feature as we begin looking to more complex datasets, as larger networks provide better non-linear pattern separation and longer memories. Additionally, it is the first time such a model has been implemented on this hardware, which was initially developed for computational neuroscience purposes.

Secondary Contributions

The contributions introduced here are preliminary in nature, and are not comprehensively explored.

1. Design considerations and architectures for NeuCube systems for Spatio-Temporal data, with particular applications in neuroinformatics.

In Appendix A, design rules for the application of SNN architectures (specifically, NeuCube architectures) to spatio-temporal data were introduced here. In particular, specific reservoir design rules for neuroinformatics data, including EEG and fMRI have been introduced. These methods are novel in SNN, and have been applied successfully in a number of studies. These studies are listed in Section A.1, and include the works by Chen et al. (2013), J. Hu et al. (2014), Kasabov, Hu et al. (2013) and Kasabov and Capecci (2015).

Additionally, in this section, some empirical evidence of the previously introduced design and implementation methodologies was given. This study showed that a NeuCube architecture developed using the concepts introduced in this thesis was effective at classifying a neuroinformatics or BCI type task. This task, the classification of EEG recordings of human motor imagery, has meaningful application in rehabilitation and medical contexts, and in general BCI.

2. Design considerations and architectures for NeuCube systems for Spectro-Temporal data, with particular applications in radioastronomy.

Some general concepts for the design of NeuCube architectures for use on spectro-temporal data are introduced here. A conceptual mapping of spectral data into a spatial format, which can then be represented in the NeuCube reservoir, was

established. This mapping retains the implicit structure of a spectro-temporal signal, which other machine learning technologies are not capable of.

Also in this section was a proof-of-concept study of a NeuCube-based SNN system's effectiveness on complex radioastronomy data, in the context of the SKA radiotelescope project. This application alone has the potential to be a significant contribution to the current state-of-the-art in both signal processing and radioastronomy itself, if the preliminary studies are any indication.

8.2 Research Questions

Considering the above contributions, and the others established in their respective chapters, we now identify if the key aims of this thesis have been met.

With regards to the Research Questions established in Section 1.1, given the contributions of this thesis we can reasonably conclude that those questions which initally motivated this work have been satisfied in their entirety. The following section will briefly discuss each of these inital questions and how they have been resolved herein. Further questions and new opportunities raised by this work have been detailed in Section 8.4.

1. Can a specific SNN framework known as the NeuCube be used to model and interpret the dynamics of a system consisting of tightly coupled spatial, spectral, and temporal data components?

This thesis has confirmed that we can indeed use the NeuCube framework – with meaningful design choices informed by the methodologies introduced here – to model and analyse systems with complex SSTD dynamics. The introduction of the design and implementation methdologies here will enforce rigour and repeatability in the developement of architectures for SNN based systems such as this.

2. Is there a design methodology we can use to inform the development of NeuCube models?

Such a design methodology has been introduced in Chapter 5, and empirical evidence of its efficacy has been shown in the case studies in this thesis. Additionally, external verification of this methodology has been shown in the studies published by other users of the NeuCube, as identified earlier in this thesis.

As those questions defined above were satisfied, the following supplementary questions were asked:

3. What considerations are there for an implementation of this system on:
   (a) Commodity computers,
   (b) Large scale clusters, or
   (c) Dedicated neuromorphic hardware?

A discussion of the considerations for these computational platforms has been given for both software, and neuromorphic hardware implementations in Chapters 6 and 7 respectively. Integrated implementations of NeuCube systems based on these considerations have been introduced for all of the named platforms.

4. Can we show some empirical evidence of a NeuCube model's effectiveness, when implemented on the above systems and designed utilising the newly established methodology, in the context of:
   (a) Spatio-Temporal data, and
   (b) Spectro-Temporal data?

Empirical evidence of NeuCube systems applied to SSTD, and developed using the methodologies and implementations developed in this thesis have been introduced. Appendix A provides evidence of this system's efficacy on spatio-temporal data in the context of a neuroinformatics or BCI task, and identifies a number of other publications showing the NeuCube's effectiveness when developed using the methodologies introduced here. Appendix B provides empirical evidence of the NeuCube's effectiveness on spectro-temporal data in the context of complex radioastronomy data, and identifies further research pathways in this area.

With these responses in mind, this thesis has satisfied its goals, and addressed those questions which initially motivated the study. These contributions go some way to establishing more rigour and repeatability in the development of SNN systems, which is typically considered to be a heuristic process with little in the way of defined methodologies. It has also introduced an effective tool for the classification and analysis of SSTD, a significant and contemporary challenge in the field of machine learning.

8.3 Caveats and Limitations of this Study

In general, the caveats and limitations of the various works in this thesis have been introduced and discussed in context, in their respective sections. Here then, we introduce only the key limitations of this study.

In a number of sections, it is identified that the decisions made are heuristically based, rather than analytically based. This is a present limitation of the existing literature, in that there is not yet a robust information theory for SNN. Here, these heuristic considerations have been made as explicit as possible, and some methodologies introduced to assist in the decision making process. However, the fact remains that the network designs we develop here are contextual. This issue is further addressed in Section 8.4.

The empirical studies introduced here are proofs-of-concept, rather than comprehensive studies. These studies provide some empirical support for the systems introduced herein. It was never the intent of this thesis to perfom large-scale, comprehensive experiments. Instead, it was intended to provide the systems and methodologies to support those empirical studies, which are in turn explored in external literature. Sufficient evidence of the efficacy of these methodologies has been established in the the contemporary literature employing them; here, we consider the publications of Capecci et al. (2015), Chen et al. (2013), Doborjeh et al. (2014a, 2014b), J. Hu et al. (2014), Kasabov and Capecci (2015), Kasabov, Hu et al. (2013), Kasabov et al. (2015), Schliebs et al. (2013), Scott et al. (2013, 2015), D. Taylor et al. (2015) and D. Taylor et al. (2014) to be evidence of their real-world application.

8.4 Open Questions & Further Work

A number of open questions have been identified in this thesis. As with the limitations of this study, these future works have primarily been identified in context, but key areas will be reintroduced here for completeness.

1. Automated optimisation of the parameters for the Threshold-Based Temporal Difference (TD) encoding scheme, using properties of the input signal; some property of the statistical variance to set the encoding threshold, and some property of the average frequency of the data (perhaps using the Shannon-Nyquist sampling theorem) to set the timestep (*cf.* Section 5.1).
2. Adoption of the Strategy design pattern for software implementation of the NeuCube (*cf.* Section 6.3.1). This would support the automated optimisation of encoding schemes and output devices.
3. The implementation of an Address-Event Representation (AER)-based communication stream to facilitate on-line applications of the NeuCube (*cf.* Section 6.3.3.1).

4. Implementation and exploration of the NeuCube Core architecture. This system is intended to address the issues of reuse and parallel development in the NeuCube, by centralising development in a modular and extensible way (*cf.* Section 6.6).
5. Further development and exploration of a NeuCube system on the Zhejiang FPGA-based neuromorphic hardware. Here we can show the NeuCube's applicability to this type of platform, which is applicable for robotics and other low-power applications (*cf.* Section 7.2).
6. An implementation of the NeuCube on INI neuromorphic VLSI devices like the cxQuad and ROLLS digital-analog ASICs. This implementation will provide a very low power consumption NeuCube model, which runs in real time. We can use such a device in a number of contexts, including neurorehabilitation, and prosthetics and robotics control. Here, we must show that the NeuCube can operate effectively with the synaptic and neuronal constraints imposed by this hardware (*cf.* Section 7.3).
7. Implementation of the streaming IO system (*cf.* Section 7.4.2.7) and subsequent adaptation of the SATA links (*cf.* Section 7.4.2.2) for high-speed streaming IO on the SpiNNaker device, for on-line applications.
8. Potential memory use reduction in the SpiNNaker device for axonal delay representation, through the development of adaptive-length ring buffers (*cf.* Section 7.4.3.1). A lower memory footprint for each synapse means that a higher number of synapses can be implemented on the device, at a lower computational cost. This would allow us to implement larger networks on the same hardware. This principle would also extend to the NeuCube Core architecture.
9. Further research on the issue of SNN systems for applications in high-speed pulsar and dispersed transient detection in radioastronomy data (*cf.* Section B.2).

Clearly, this research has identified a number of valuable areas for further work. This is, of course, outside of the main area of further work identified here – that of a robust theory for SNN systems. To this end, we intend to begin a study on the sensistivity of such systems (initially using the Latin Hypercube method) to their structure and parameters, to provide further empirical support to the methodologies introduced in this thesis. Additionally, we intend to approch the behaviour of these networks from an information theory perspective. Interesting studies using a similar method have been performed on biological neurons in Rolls and Treves (2011) and Quian Quiroga and Panzeri (2009), and these methods can logically be applied to

artifical neurons. This concept has not been discussed in this thesis, but there is some overlap between the two; with a better understanding of the properties of such networks, we can make more informed design decisions – and, perhaps even replace the heuristics with analytically optimal solutions.

8.5 Closing Thoughts

I began this thesis with a quote from the late J.Z. Young, wherein he predicts that '...the principles now being discovered in the brain may provide, in the future, machines even more powerful than those we can at present forsee.' It is an innocuous quote, perhaps obvious with hindsight, though I feel it is no less meaningful for its simplicity.

We now find ourselves five and a half decades into that brave future; more than a half-century on from the birth of neural network modelling in the seminal works of McCulloch, Minsky, Pitts, Rosenblatt, and Werbos; more than twenty years since the start of the modern neural network revolution driven by Izhikevich, Mead, Maass, and Thorpe; in the midst of an unprecedented interest in biologically-inspired and biomimetic computation; at a moment in time where the massive computational demand of such systems can finally begin to be satiated; perhaps at the start of another renaissance in neurally inspired computing; certainly in a window of immense opportunity. It is this future in which we see the 'powerful machines' envisioned generations ago rapidly becoming a reality.

In my opinion, it is inevitable that computational intelligence will eventually converge on brain-like methods of adaptation and learning. The human brain is too powerful a draw – with its adapation, capacity for depth and breadth of learning, and extreme energy efficiency, it is the ideal blueprint from which to draw inspiration for the computational systems intended to demonstrate these same properties. In this, I do not imply that our systems will reproduce the specific dynamics of human brain in a biologically accurate manner. Personally, I do not believe this to the be the optimal solution. The abstraction of these dynamics have remarkable emergent properties of their own. In conjunction with the immensely powerful computational platforms currently in development, they may even show better results than our biological 'computers'.

The NeuCube, and its derivatives, are one such 'powerful machine'. Herein, I have demonstrated its efficacy on disparate platforms and datasets, and indeed, it has shown excellent promise. Is this system a step on the road towards something greater? Almost certainly, if we look to a long enough time scale. Regardless, in the

here and now, I contend that the NeuCube and the systems introduced in this thesis currently provide one of the most effective techniques for addressing complex and challenging SSTD.

References

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

Adrian, E. D. (1926). The impulses produced by sensory nerve-endings. *Journal of Physiology*, *61*(1), 49–72.

Aggarwal, V., Acharya, S., Tenore, F., Hyun-Chool, S., Etienne-Cummings, R., Schieber, M. & Thankor, N. (2008). Asynchronous Decoding of Dexterous Finger Movements Using M1 Neurons. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, *16*(1), 3–14.

Alexander, A. L., Lee, J. E., Lazar, M. & Field, A. S. (2007, July). Diffusion Tensor Imaging of the Brain. *Neurotherapeutics*, *4*(3), 316–329. doi:10.1016/j.nurt.2007.05.011

Allen, C., Celikel, T. & Feldman, D. (2003, March). Long-term depression induced by sensory deprivation during cortical map plasticity in vivo. *Nature Neuroscience*, *6*(3), 291–9. doi:10.1038/nn1012

Allen, C. & Stevens, C. F. (1994, October). An evaluation of causes for unreliability of synaptic transmission. *Proceedings of the National Academy of Sciences of the United States of America*, *91*(22), 10380–3.

Amit, D. J. & Fusi, S. (1994, September). Learning in Neural Networks with Material Synapses. *Neural Computation*, *6*(5), 957–982. doi:10.1162/neco.1994.6.5.957

Andrienko, G., Malerba, D., May, M. & Teisseire, M. (2006, November). Mining spatio-temporal data. *Journal of Intelligent Information Systems*, *27*(3), 187–190. doi:10.1007/s10844-006-9949-3

Angélil, R., Saha, P. & Merritt, D. (2010, September). Toward Relativistic Orbit Fitting of Galactic Center Stars and Pulsars. *The Astrophysical Journal*, *720*(2), 1303–1310. doi:10.1088/0004-637X/720/2/1303

Antoni, J. (2006, February). The spectral kurtosis: a useful tool for characterising non-stationary signals. *Mechanical Systems and Signal Processing*, *20*(2), 282–307. doi:10.1016/j.ymssp.2004.09.001

Arvaneh, M., Guan, C., Ang, K. K. & Quek, C. (2011). Optimizing the channel selection and classification accuracy in EEG-based BCI. *IEEE Transactions on Biomedical Engineering*, *58*(6), 1865–1873. doi:10.1109/TBME.2011.2131142

Aurlien, H., Gjerde, I. O., Aarseth, J. H., Eldøen, G., Karlsen, B., Skeidsvoll, H. & Gilhus, N. E. (2004, March). EEG background activity described by a large computerized database. *Clinical Neurophysiology*, *115*(3), 665–673. doi:10.1016/j.clinph.2003.10.019

Azadmehr, M., Abrahamsen, J. P. & Häfliger, P. (2005). A foveated AER imager chip. In *2005 ieee international symposium on circuits and systems* (Vol. 1, pp. 2751–2754). Kobe, Japan: IEEE. doi:10.1109/ISCAS.2005.1465196

Azghadi, M. R., Iannella, N., Al-Sarawi, S. F., Indiveri, G. & Abbott, D. (2014, May). Spike-Based Synaptic Plasticity in Silicon: Design, Implementation, Application, and Challenges. *Proceedings of the IEEE*, *102*(5), 717–737. doi:10.1109/JPROC.2014.2314454

Backer, D. C. & Hellings, R. W. (1986, September). Pulsar Timing and General Relativity. *Annual Review of Astronomy and Astrophysics*, *24*(1), 537–575. doi:10.1146/annurev.aa.24.090186.002541

Backer, D. C., Kulkarni, S. R., Heiles, C., Davis, M. M. & Goss, W. M. (1982, December). A Millisecond Pulsar. *Nature*, *300*(5893), 615–618. doi:10.1038/300615a0

Badoni, D., Giulioni, M., Dante, V. & Del Giudice, P. (2006). An aVLSI recurrent network of spiking neurons with reconfigurable and plastic synapses. In *Proceedings of the ieee international symposium on circuits and systems* (p. 4). Island of Kos, Greece: IEEE. doi:10.1109/ISCAS.2006.1692813

Bear, M., Connors, B. & Paradiso, M. (2007). *Neuroscience: Exploring the Brain* (3rd). Phildelphia, PA, USA: Lippincott Williams & Wilkins.

Belatreche, A., Maguire, L. P. & McGinnity, M. (2006, March). Advances in Design and Application of Spiking Neural Networks. *Soft Computing*, *11*(3), 239–248. doi:10.1007/s00500-006-0065-7

Bengio, Y. & LeCun, Y. (2007). Scaling Learning Algorithms towards AI. *Large Scale Kernel Machines*, (1), 321–360.

Bentley, J. L. (1975, September). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, *18*(9), 509–517. doi:10.1145/361002.361007

Benuskova, L. & Kasabov, N. (2008, December). Modeling brain dynamics using computational neurogenetic approach. *Cognitive Neurodynamics*, *2*(4), 319–34. doi:10.1007/s11571-008-9061-1

Bi, G.-Q. & Poo, M.-M. (1998). Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type. *The Journal of Neuroscience*, *18*(24), 10464–10472.

Bi, G.-Q. & Poo, M.-M. (2001, January). Synaptic modification by correlated activity: Hebb's postulate revisited. *Annual Review of Neuroscience*, *24*, 139–66. doi:10.1146/annurev.neuro.24.1.139

Bogorny, V. & Shekhar, S. (2010, December). Spatial and Spatio-temporal Data Mining. In *2010 ieee international conference on data mining* (pp. 1217–1217). IEEE. doi:10.1109/ICDM.2010.166

Bohte, S. (2004). The evidence for neural information processing with precise spike-times: A survey. *Natural Computing*, *3*(2), 195–206. doi:10.1023/B:NACO.0000027755.02868.60

Bohte, S., Kok, J. & Poutre, J. (2002, October). Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons. *Neurocomputing*, *48*(1-4), 17–37. doi:10.1016/S0925-2312(01)00658-0

Brader, J. M., Senn, W. & Fusi, S. (2007, November). Learning real-world stimuli in a neural network with spike-driven synaptic dynamics. *Neural Computation*, *19*(11), 2881–912. doi:10.1162/neco.2007.19.11.2881

Brette, R. & Gerstner, W. (2005). Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity. *Journal of Neurophysiology*, *94*(5), 3637–3642. doi:10.1152/jn.00686.2005.

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., … Destexhe, A. (2007, December). Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of Computational Neuroscience*, *23*(3), 349–98. doi:10.1007/s10827-007-0038-6

Bruckner, S., Solteszova, V., Groller, E., Hladuvka, J., Buhler, K., Yu, J. Y., … Dickson, B. J. (2009, November). BrainGazer - Visual Queries for Neurobiology Research. *IEEE Transactions on Visualization and Computer Graphics*, *15*(6), 1497–1504. doi:10.1109/TVCG.2009.121

Brunel, N. & Van Rossum, M. (2007, December). Lapicque's 1907 paper: from frogs to integrate-and-fire. *Biological Cybernetics*, *97*(5-6), 337–339. doi:10.1007/s00422-007-0190-0

Buteneers, P., Schrauwen, B., Verstraeten, D. & Stroobandt, D. (2008). Real-time epileptic seizure detection on intra-cranial rat data using reservoir computing. In *Proceedings of the international conference on neural information processing* (pp. 56–63). Auckland, New Zealand: Springer. doi:10.1007/978-3-642-02490-0_7

Butts, D. A., Weng, C., Jin, J., Yeh, C.-I., Lesica, N. A., Alonso, J.-M. & Stanley, G. B. (2007, September). Temporal precision in the neural code and the timescales of natural vision. *Nature*, *449*(7158), 92–5. doi:10.1038/nature06105

Capecci, E., Kasabov, N. & Wang, G. (2015, August). Analysis of connectivity in NeuCube spiking neural network models trained on EEG data for the understanding of functional changes in the brain: A case study on opiate dependence treatment. *Neural Networks*, *68*, 62–77. doi:10.1016/j.neunet.2015.03.009

Cassidy, A., Denham, S., Kanold, P. & Andreou, A. (2007, November). FPGA Based Silicon Spiking Neural Array. In *Proceedings of the ieee biomedical circuits and systems conference* (pp. 75–78). Montreal, Quebec, Canada: IEEE. doi:10.1109/BIOCAS.2007.4463312

Chang, S. H., Zhou, P., Rymer, W. Z. & Li, S. (2013). Spasticity, weakness, force variability, and sustained spontaneous motor unit discharges of resting spastic–paretic biceps brachii muscles in chronic stroke. *Muscle & Nerve*, *48*(1), 85–92. doi:10.1002/mus.23699

Chechik, G. (2003, July). Spike-timing-dependent plasticity and relevant mutual information maximization. *Neural Computation*, *15*(7), 1481–510. doi:10.1162/089976603321891774

Chen, Y., Hu, J., Kasabov, N., Hou, Z.-G. & Cheng, L. (2013). NeuCubeRehab: A Pilot Study for EEG Classification in Rehabilitation Practice Based on Spiking Neural Networks. In *Proceedings of the international conference on neural information processing* (pp. 70–77). Springer. doi:10.1007/978-3-642-42051-1_10

Chestek, C. & Shenoy, K. (2012). Neural Prosthetics. *Scholarpedia*, *7*(3), 11854. doi:10.4249/scholarpedia.11854

Chua, L. (1971). Memristor-The missing circuit element. *IEEE Transactions on Circuit Theory*, *18*(5), 507–519. doi:10.1109/TCT.1971.1083337

Chua, L. (2011, March). Resistance switching memories are memristors. *Applied Physics A*, *102*(4), 765–783. doi:10.1007/s00339-011-6264-9

Cortes, C. & Vapnik, V. N. (1995). Support Vector Networks. *Machine Learning*, *20*, 273–297.

Cox, D. D. & Savoy, R. L. (2003, June). Functional magnetic resonance imaging (fMRI) "brain reading": detecting and classifying distributed patterns of fMRI activity in human visual cortex. *NeuroImage*, *19*(2), 261–270. doi:10.1016/S1053-8119(03)00049-1

Davies, S., Galluppi, F., Rast, A. & Furber, S. (2012, August). A forecast-based STDP rule suitable for neuromorphic implementation. *Neural Networks*, *32*, 3–14. doi:10.1016/j.neunet.2012.02.018

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., … Yger, P. (2008, January). PyNN: A Common Interface for Neuronal Network Simulators. *Frontiers in Neuroinformatics*, *2*, 11. doi:10.3389/neuro.11.011.2008

Dayan, P. & Abbott, L. F. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Cambridge, MA, USA: MIT Press.

de Garis, H., Korkin, M. & Fehr, G. (2001). The CAM-Brain Machine (CBM): An FPGA Based Tool for Evolving a 75 Million Neuron Artificial Brain to Control a Lifesized Kitten Robot. *Autonomous Robots*, *10*(3), 235–249. doi:10.1023/A:1011286308522

Debanne, D., Campanac, E., Bialowas, A., Carlier, E. & Alcaraz, G. (2011). Axon Physiology. *Physiological Reviews*, *91*(2), 555–602. doi:10.1152/physrev.00048.2009

Deep, K., Singh, K. P., Kansal, M. L. & Mohan, C. (2009). A real coded genetic algorithm for solving integer and mixed integer optimization problems. *Applied Mathematics and Computation*, *212*(2), 505–518. doi:10.1016/j.amc.2009.02.044

Defoin-Platel, M., Schliebs, S. & Kasabov, N. (2009). Quantum-inspired Evolutionary Algorithm: A multi-model EDA. *IEEE Trans. Evolutionary Computation*, *13*(6), 1218–32.

Delbruck, T. & Lichtsteiner, P. (2007, May). Fast sensory motor control based on event-based hybrid neuromorphic-procedural system. In *2007 ieee international symposium on circuits and systems* (pp. 845–848). New Orleans, LA, USA: IEEE. doi:10.1109/ISCAS.2007.378038

Deneva, J. S., Cordes, J. M. & Lazio, T. J. W. (2009, September). Discovery of Three Pulsars from a Galactic Center Pulsar Population. *The Astrophysical Journal*, *702*(2), L177–L181. doi:10.1088/0004-637X/702/2/L177

Desbordes, G., Jin, J., Alonso, J.-M. & Stanley, G. B. (2010, January). Modulation of temporal precision in thalamic population responses to natural visual stimuli. *Frontiers in Systems Neuroscience*, *4*, 151. doi:10.3389/fnsys.2010.00151

Desbordes, G., Jin, J., Weng, C., Lesica, N. A., Stanley, G. B. & Alonso, J.-M. (2008, December). Timing precision in population coding of natural scenes in the early visual system. *PLoS Biology*, *6*(12), e324. doi:10.1371/journal.pbio.0060324

Dewdney, P., Turner, W., Braun, R., Juande, S.-V. & Tan, G. H. (2015). *SKA1 System Baseline V2 Description*. Square Kilometre Array. Manchester, UK. Retrieved from https://www.skatelescope.org/key-documents/

Dhoble, K., Nuntalid, N., Indiveri, G. & Kasabov, N. (2012, June). Online spatio-temporal pattern recognition with evolving spiking neural networks utilising address event representation, rank order, and temporal spike learning. In *Proceedings of the 2012 international joint conference on neural networks* (pp. 1–7). Brisbane, Australia: IEEE. doi:10.1109/IJCNN.2012.6252439

Diehl, P. U. & Cook, M. (2014, July). Efficient implementation of STDP rules on SpiNNaker neuromorphic hardware. In *2014 international joint conference on neural networks (ijcnn)* (pp. 4288–4295). Beijing, China: IEEE. doi:10.1109/IJCNN.2014.6889876

Diesmann, M. & Gewaltig, M.-O. (2001). NEST: An environment for neural systems simulation. *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis*, *58*, 43–70.

Djurfeldt, M. (2012). The connection-set algebra-a novel formalism for the representation of connectivity structure in neuronal network models. *Neuroinformatics*, *10*(3), 287–304. doi:10.1007/s12021-012-9146-1

Doborjeh, M. G., Capecci, E. & Kasabov, N. (2014a, December). Classification and segmentation of fMRI Spatio-Temporal Brain Data with a NeuCube evolving Spiking Neural Network model. In *2014 ieee symposium on evolving and autonomous learning systems (eals)* (pp. 73–80). Orlando, FL, USA: IEEE. doi:10.1109/EALS.2014.7009506

Doborjeh, M. G., Capecci, E. & Kasabov, N. (2014b). Temporal Brain Data with a NeuCube Evolving Spiking Neural Network Model, 73–80.

Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K. & Burger, D. (2011, July). Dark silicon and the end of multicore scaling. *ACM SIGARCH Computer Architecture News*, *39*(3), 365. doi:10.1145/2024723.2000108

Esser, S. K., Andreopoulos, A., Appuswamy, R., Datta, P., Barch, D., Amir, A., … Modha, D. S. (2013, August). Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores. In *Proceedings of the 2013 international joint conference on neural networks* (pp. 1–10). Dallas, TX, USA: IEEE. doi:10.1109/IJCNN.2013.6706746

Evarts, E. V. (1968, January). Relation of pyramidal tract activity to force exerted during voluntary movement. *Journal of Neurophysiology*, *31*(1), 14–27. Retrieved from http://www.ncbi.nlm.nih.gov/pubmed/4966614

Fadiga, L., Fogassi, L., Pavesi, G. & Rizzolatti, G. (1995, June). Motor facilitation during action observation: a magnetic stimulation study. *Journal of Neurophysiology*, *73*(6), 2608–2611. Retrieved from http://jn.physiology.org/content/73/6/2608

Fagg, A., Hatsopoulos, N. G., de Lafuente, V., Moxon, K., Nemati, S., Rebesco, J., … Miller, L. (2007). Biomimetic Brain Machine Interfaces for the Control of Movement. *The Journal of Neuroscience*, *27*(44), 11842–11846. doi:10.1523/JNEUROSCI.3516-07.2007

Feng, W.-C. (2003, October). Making a Case for Efficient Supercomputing. *Queue*, *1*(7), 54. doi:10.1145/957717.957772

Florian, R. V. (2008). Tempotron-like learning with ReSuMe. In V. Kurkova, R. Neruda & J. Koutnik (Eds.), *Proceedings of the 18th international conference on artificial neural networks* (Vol. 5164, pp. 368–375). Lecture Notes in Computer Science. Prague, Czech Republic: Springer Berlin Heidelberg. doi:10.1007/978-3-540-87559-8_38

Florian, R. V. (2010). The Chronotron : a neuron that learns to fire temporally-precise spike patterns. *Nature Preceedings*. doi:10.1038/npre.2010.5190.3

Florian, R. V. (2012, August). The Chronotron: A Neuron That Learns to Fire Temporally Precise Spike Patterns. *PLoS ONE*, *7*(8), e40233. doi:10.1371/journal.pone.0040233

Foottit, J., Brown, D., Marks, S. & Connor, A. M. (2014). An Intuitive Tangible Game Controller. In *Proceedings of the 2014 conference on interactive entertainment - ie2014* (pp. 1–7). New York, New York, USA: ACM Press. doi:10.1145/2677758.2677774

Formisano, E., de Martino, F. & Valente, G. (2008, September). Multivariate analysis of fMRI time series: classification and regression of brain responses using machine learning. *Magnetic Resonance Imaging*, *26*(7), 921–34. doi:10.1016/j.mri.2008.01.052

Fox, K. (2009). Experience-dependent plasticity mechanisms for neural rehabilitation in somatosensory cortex. *Philosophical Transactions of the Royal Society of London. Series B, Biological sciences*, *364*(1515), 369–381. doi:10.1098/rstb.2008.0252

Fransson, P., Krüger, G., Merboldt, K. D. & Frahm, J. (1999). *MRI of functional deactivation: temporal and spatial characteristics of oxygenation-sensitive responses in human visual cortex*. doi:10.1006/nimg.1999.0438

Freeman, D. K., Rizzo, J. F. & Fried, S. I. (2011, June). Encoding visual information in retinal ganglion cells with prosthetic stimulation. *Journal of Neural Engineering*, *8*(3), 035005. doi:10.1088/1741-2560/8/3/035005

Frégnac, Y., Pananceau, M., René, A., Huguet, N., Marre, O., Levy, M. & Shulz, D. (2010, January). A Re-Examination of Hebbian-Covariance Rules and Spike Timing-Dependent Plasticity in Cat Visual Cortex in vivo. *Frontiers in Synaptic Neuroscience*, *2*, 147. doi:10.3389/fnsyn.2010.00147

Furber, S. (2012, August). To Build a Brain. *IEEE Spectrum*, *49*(8), 44–49. doi:10.1109/MSPEC.2012.6247562

Furber, S. (2016). Large-scale neuromorphic computing systems. *Journal of Neural Engineering*, *13*(5), 051001. doi:10.1088/1741-2560/13/5/051001

Furber, S., Lester, D., Plana, L. A., Garside, J. D., Painkras, E., Temple, S. & Brown, A. D. (2013, December). Overview of the SpiNNaker System Architecture. *IEEE Transactions on Computers*, *62*(12), 2454–2467. doi:10.1109/TC.2012.142

Fusi, S. (2000). Spike-Driven Synaptic Plasticity: Theory, Simulation, VLSI Implementation. *Neural Computation*, *12*, 2227–2258.

Fusi, S. (2003, January). Spike-driven synaptic plasticity for learning correlated patterns of mean firing rates. *Reviews in the Neurosciences*, *14*(1-2), 73–84.

Fusi, S., Drew, P. J. & Abbott, L. F. (2005, February). Cascade models of synaptically stored memories. *Neuron*, *45*(4), 599–611. doi:10.1016/j.neuron.2005.02.001

Galluppi, F., Lagorce, X., Stromatias, E., Pfeiffer, M., Plana, L. A., Furber, S. & Benosman, R. B. (2015, January). A framework for plasticity implementation on the SpiNNaker neural architecture. *Frontiers in Neuroscience*, *8*. doi:10.3389/fnins.2014.00429

Galluppi, F., Rast, A., Davies, S. & Furber, S. (2010). A general-purpose model translation system for a universal neural chip. In *Proceedings of the international conference on neural information processing* (pp. 58–65). Sydney, Australia.

Gamma, E., Helm, R., Johnson, R. E. & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. doi:10.1093/carcin/bgs084

Garcia, G. N., Ebrahimi, T. & Vesin, J. M. (2003). Support vector EEG classification in the Fourier and time-frequency correlation domains. In *First international ieee embs conference on neural engineering* (pp. 591–594). IEEE. doi:10.1109/CNE.2003.1196897

Gelencsér, A., Prodromakis, T., Toumazou, C. & Roska, T. (2012, April). Biomimetic model of the outer plexiform layer by incorporating memristive devices. *Physical Review E*, *85*(4), 041918. doi:10.1103/PhysRevE.85.041918

Georgopoulos, A. P., Kalaska, J. F., Caminiti, R. & Massey, J. T. (1982, November). On the relations between the direction of two-dimensional arm movements and cell discharge in primate motor cortex. *The Journal of Neuroscience*, *2*(11), 1527–37.

Gerstner, W. (2010, January). From Hebb Rules to Spike-Timing-Dependent Plasticity: A Personal Account. *Frontiers in Synaptic Neuroscience*, *2*, 151. doi:10.3389/fnsyn.2010.00151

Gerstner, W. & Brette, R. (2009). Adaptive exponential integrate-and-fire model. *Scholarpedia*, *4*(6), 8427. doi:10.4249/scholarpedia.8427

Gerstner, W. & Kistler, W. (2002). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge, MA, USA: Cambridge University Press.

Gerstner, W., Kistler, W., Naud, R. & Paninski, L. (2014). *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition* (1st). Cambridge, UK: Cambridge University Press.

Gewaltig, M.-O. & Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia*, *2*(4), 1430. doi:10.4249/scholarpedia.1430

Ghosh-Dastidar, S. & Adeli, H. (2007). Improved spiking neural networks for EEG classification and epilepsy and seizure detection. *Integrated Computer-Aided Engineering*, *14*, 187–212.

Ghosh-Dastidar, S. & Adeli, H. (2009, December). A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection. *Neural Networks*, *22*(10), 1419–31. doi:10.1016/j.neunet.2009.04.003

Glover, G. H. (1999). Deconvolution of impulse response in event-related BOLD fMRI. *NeuroImage*, *9*(4), 416–429. doi:10.1006/nimg.1998.0419

Goodman, D. & Brette, R. (2008, January). Brian: a simulator for spiking neural networks in Python. *Frontiers in Neuroinformatics*, *2*, 5. doi:10.3389/neuro.11.005.2008

Goudarzi, A., Banda, P., Lakin, M. R., Teuscher, C. & Stefanovic, D. (2014, January). *A Comparative Study of Reservoir Computing for Temporal Signal Processing*. arXiv: 1401.2224. Retrieved from http://arxiv.org/abs/1401.2224

Graas, E. L., Brown, E. A. & Lee, R. H. (2004). An FPGA-Based Approach to High-Speed Simulation of Conductance-Based Neuron Models. *Neuroinformatics*, *2*(4), 417–435.

Gray, V., Rice, C. L. & Garland, S. J. (2012). Factors that influence muscle weakness following stroke and their clinical implications: a critical review. *Physiotherapy Canada*, *64*(4), 415–426. doi:10.3138/ptc.2011-03

Grush, R. (2004, June). The emulation theory of representation: motor control, imagery, and perception. *The Behavioral and Brain Sciences*, *27*(3), 377–96. Retrieved from http://www.ncbi.nlm.nih.gov/pubmed/15736871

Gütig, R., Aharonov, R., Rotter, S. & Sompolinsky, H. (2003). Learning input correlations through nonlinear temporally asymmetric Hebbian plasticity. *The Journal of Neuroscience*, *23*, 3697–3714.

Gütig, R. & Sompolinsky, H. (2006, March). The tempotron: a neuron that learns spike timing-based decisions. *Nature Neuroscience*, *9*(3), 420–8. doi:10.1038/nn1643

Hardingham, N. R. & Larkman, A. U. (1998, February). Rapid report: the reliability of excitatory synaptic transmission in slices of rat visual cortex in vitro is temperature dependent. *The Journal of Physiology*, *507*, 249–56.

Hawrylycz, M. J., Lein, E. S., Guillozet-Bongaarts, A. L., Shen, E. H., Ng, L., Miller, J. A., … Jones, A. R. (2012, September). An anatomically comprehensive atlas of the adult human brain transcriptome. *Nature*, *489*(7416), 391–9. doi:10.1038/nature11405

Hazan, H. & Manevitz, L. M. (2012, February). Topological constraints and robustness in liquid state machines. *Expert Systems with Applications*, *39*(2), 1597–1606. doi:10.1016/j.eswa.2011.06.052

Hebb, D. (1949). *The Organization of Behavior*. New York, NY, USA: Wiley.

Heeger, D. J. & Ress, D. (2002, February). What does fMRI tell us about Neuronal Activity? *Nature Reviews Neuroscience*, *3*(2), 142–151. doi:10.1038/nrn730

Herbert, J. H. (1958, May). Report of the committee on methods of clinical examination in electroencephalography. *Electroencephalography and Clinical Neurophysiology*, *10*(2), 370–375. doi:10.1016/0013-4694(58)90053-1

Hines, M. & Carnevale, T. (2001, April). NEURON: a tool for neuroscientists. *The Neuroscientist*, *7*(2), 123–35.

Hinton, G. E., Osindero, S. & Teh, Y.-W. (2006, July). A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, *18*(7), 1527–1554. doi:10.1162/neco.2006.18.7.1527

Hochberg, L. R., Serruya, M. D., Friehs, G. M., Mukand, J. A., Saleh, M., Caplan, A. H., … Donoghue, J. P. (2006, July). Neuronal ensemble control of prosthetic devices by a human with tetraplegia. *Nature*, *442*(7099), 164–71. doi:10.1038/nature04970

Hodgkin, A. & Huxley, A. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, *117*(4), 500–544.

Holper, L. & Wolf, M. (2011). Single-trial classification of motor imagery differing in task complexity: a functional near-infrared spectroscopy study. *Journal of Neuroengineering and Rehabilitation*, *8*, 34. doi:10.1186/1743-0003-8-34

Hopkins, M. & Furber, S. (2015, October). Accuracy and Efficiency in Fixed-Point Neural ODE Solvers. *Neural Computation*, *27*(10), 2148–2182. doi:10.1162/NECO_a_00772

Hough, M., de Garis, H., Korkin, M., Gers, F. & Nawa, N. E. (1999). SPIKER Analog waveform to digital spiketrain conversion in ATR's artificial brain (CAM-Brain) project. In *International conference on robotics and artificial life*.

Hu, J., Hou, Z.-G., Chen, Y., Kasabov, N. & Scott, N. M. (2014, August). EEG-based classification of upper-limb ADL using SNN for active robotic rehabilitation. In *5th ieee ras/embs international conference on biomedical robotics and biomechatronics* (pp. 409–414). IEEE. doi:10.1109/BIOROB.2014.6913811

Hu, M., Li, H., Chen, Y., Wang, X. & Pino, R. E. (2011, January). Geometry variations analysis of TiO<inf>2</inf> thin-film and spintronic memristors. In *16th asia and south pacific design automation conference (asp-dac 2011)* (pp. 25–30). Yokohama, Japan: IEEE. doi:10.1109/ASPDAC.2011.5722193

Iakymchuk, T., Rosado, A., Serrano-Gotarredona, T., Linares-Barranco, B., Jimenez-Fernandez, A., Linares-Barranco, A. & Jimenez-Moreno, G. (2014, June). An AER handshake-less modular infrastructure PCB with x8 2.5Gbps LVDS serial links. In *2014 ieee international symposium on circuits and systems (iscas)* (pp. 1556–1559). Melboure, Australia: IEEE. doi:10.1109/ISCAS.2014.6865445

Indiveri, G. (2003). A low-power adaptive integrate-and-fire neuron circuit. In *Proceedings of the 2003 international symposium on circuits and systems* (Vol. 4, pp. IV–820–IV–823). Bangkok, Thailand: IEEE. doi:10.1109/ISCAS.2003.1206342

Indiveri, G., Corradi, F. & Qiao, N. (2015). Neuromorphic Architectures for Spiking Deep Neural Networks. In *Proceedings of the international electron devices meeting* (pp. 1–4). Washington DC, USA: IEEE.

Indiveri, G. & Horiuchi, T. K. (2011, January). Frontiers in neuromorphic engineering. *Frontiers in Neuroscience*, *5*, 118. doi:10.3389/fnins.2011.00118

Indiveri, G., Linares-Barranco, B., Hamilton, T. J., van Schaik, A., Etienne-Cummings, R., Delbruck, T., … Boahen, K. (2011, January). Neuromorphic silicon neuron circuits. *Frontiers in Neuroscience*, *5*, 73. doi:10.3389/fnins.2011.00073

Indiveri, G., Linares-Barranco, B., Legenstein, R., Deligeorgis, G. & Prodromakis, T. (2013, September). Integration of nanoscale memristor synapses in neuromorphic computing architectures. *Nanotechnology*, *24*(38), 384010. doi:10.1088/0957-4484/24/38/384010

Indiveri, G., Stefanini, F. & Chicca, E. (2010, May). Spike-based learning with a generalized integrate and fire silicon neuron. In *Proceedings of 2010 ieee international symposium on circuits and systems* (pp. 1951–1954). Paris, France: IEEE. doi:10.1109/ISCAS.2010.5536980

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, *14*(6), 1569–1572. doi:10.1109/TNN.2003.820440

Izhikevich, E. M. (2004). Which Model to Use for Cortical Spiking Neurons? *IEEE Transactions on Neural Networks*, *15*(5), 1063–1070. doi:10.1109/TNN.2004.832719

Jaeger, H. (2001). *The "echo state"approach to analysing and training recurrent neural networks*, Fraunhofer Institute for Autonomous Intelligent Systems. St. Augustin. Retrieved from http://minds.jacobs-university.de/sites/default/files/uploads/papers/EchoStatesTechRep.pdf

Jaeger, H. (2002). *Short term memory in echo state networks*, Fraunhofer Institute for Autonomous Intelligent Systems. St. Augustin. Retrieved from http://minds.jacobs-university.de/sites/default/files/uploads/papers/STMEchoStatesTechRep.pdf

Jaeger, H. (2007). Echo state network. *Scholarpedia*, *2*(9), 2330. doi:10.4249/scholarpedia.2330

Jeannerod, M. (1994). The representing brain: Neural correlates of motor intention and imagery. *Behavioral and Brain Sciences*, *17*(02), 187. doi:10.1017/S0140525X00034026

Jeannerod, M. (2001). Neural simulation of action: A unifying mechanism for motor cognition. *NeuroImage*, *14*, S103–S109. doi:10.1006/nimg.2001.0832

Jin, X., Furber, S. & Woods, J. V. (2008, June). Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In *2008 ieee international joint conference on neural networks* (pp. 2812–2819). Hong Kong, China: IEEE. doi:10.1109/IJCNN.2008.4634194

Jin, X., Rast, A., Galluppi, F., Davies, S. & Furber, S. (2010, July). Implementing spike-timing-dependent plasticity on SpiNNaker neuromorphic hardware. In *Proceedings of the international joint conference on neural networks* (pp. 1–8). Barcelona, Spain: Ieee. doi:10.1109/IJCNN.2010.5596372

Jo, S. H., Chang, T., Ebong, I., Bhadviya, B. B., Mazumder, P. & Lu, W. (2010, April). Nanoscale Memristor Device as Synapse in Neuromorphic Systems. *Nano Letters*, *10*(4), 1297–1301. doi:10.1021/nl904092h

Jolivet, R., Lewis, T. J. & Gerstner, W. (2003). The Spike Response Model: A Framework to Predict Neuronal Spike Trains. In R. Jolivet, J. Timothy & W. Gerstner (Eds.), *Proceedings of the 2003 joint international conference icann/iconip* (pp. 846–853). Istanbul, Turkey: Springer Berlin Heidelberg. doi:10.1007/3-540-44989-2_101

Kandel, E. & Schwartz, J. (Eds.). (2000). *Principles of Neural Science* (4th). New York, NY, USA: McGraw-Hill.

Kasabov, N. (1998). Evolving Fuzzy Neural Networks - Algorithms, Applications and Biological Motivation. In T. Yamakawa & G. Matsumoto (Eds.), *World scientific*.

Kasabov, N. (2007). *Evolving Connectionist Systems: The Knowledge Engineering Approach*. Springer.

Kasabov, N. (2010, January). To Spike or not to Spike: A Probabilistic Spiking Neuron Model. *Neural Networks*, *23*(1), 16–9. doi:10.1016/j.neunet.2009.08.010

Kasabov, N. (2012a). Evolving, Probabilistic Spiking Neural Networks and Neurogenetic Systems for Spatio- and Spectro-Temporal Data Modelling and Pattern Recognition. *Natural Intelligence*. Lecture Notes in Computer Science, *1*(2), 234–260. doi:10.1007/978-3-642-30687-7_12

Kasabov, N. (2012b). NeuCube EvoSpike Architecture for Spatio-temporal Modelling and Pattern Recognition of Brain Signals. In A. Mana, F. Schwenker & E. Trentin (Eds.), *Lncs 7477* (pp. 225–243). Trento, Italy: Springer. doi:10.1007/978-3-642-33212-8_21

Kasabov, N. (2014, April). NeuCube: A spiking neural network architecture for mapping, learning and understanding of spatio-temporal brain data. *Neural Networks*, *52*, 62–76. doi:10.1016/j.neunet.2014.01.006

Kasabov, N. (2015, May). Evolving connectionist systems for adaptive learning and knowledge discovery: Trends and directions. *Knowledge-Based Systems*, *80*, 24–33. doi:10.1016/j.knosys.2014.12.032

Kasabov, N., Benuskova, L. & Wysoski, S. G. (2005a). A computational neurogenetic model of a spiking neuron. *Proceedings of the IEEE International Joint Conference on Neural Networks*, *2*, 446–451. doi:10.1109/IJCNN.2005.1555872

Kasabov, N., Benuskova, L. & Wysoski, S. G. (2005b, December). Biologically Plausible Computational Neurogenetic Models: Modeling the Interaction Between Genes, Neurons and Neural Networks. *Journal of Computational and Theoretical Nanoscience*, *2*(4), 569–573. doi:10.1166/jctn.2005.012

Kasabov, N., Benuskova, L. & Wysoski, S. G. (2005c, December). Computational Neurogenetic Modeling: Integration of Spiking Neural Networks, Gene Networks, and Signal Processing Techniques. In *Proceedings of the international conference on artificial neural networks* (pp. 509–514). Warsaw, Poland: Springer. doi:10.1007/11550907_80

Kasabov, N. & Capecci, E. (2015, February). Spiking neural network methodology for modelling, classification and understanding of EEG spatio-temporal data measuring cognitive processes. *Information Sciences*, *294*, 565–575. doi:10.1016/j.ins.2014.06.028

Kasabov, N., Dhoble, K., Nuntalid, N. & Indiveri, G. (2013). Dynamic evolving spiking neural networks for on-line spatio- and spectro-temporal pattern recognition. *Neural Networks*, *41*(1995), 188–201. doi:10.1016/j.neunet.2012.11.014

Kasabov, N., Hu, J., Chen, Y., Scott, N. M. & Turkova, Y. (2013). Spatio-temporal EEG Data Classification in the NeuCube 3D SNN Environment: Methodology and Examples. In *20th international conference on neural information processing* (pp. 63–69). Daegu, Korea: Springer. doi:10.1007/978-3-642-42051-1_9

Kasabov, N., Schliebs, R. & Kojima, H. (2011, December). Probabilistic Computational Neurogenetic Modeling: From Cognitive Systems to Alzheimer's Disease. *IEEE Transactions on Autonomous Mental Development*, *3*(4), 300–311. doi:10.1109/TAMD.2011.2159839

Kasabov, N., Scott, N. M., Tu, E., Marks, S., Sengupta, N., Capecci, E., … Yang, J. (2015). Evolving spatio-temporal data machines based on the NeuCube neuromorphic framework: design methodology and selected applications. *Neural Networks*. doi:10.1016/j.neunet.2015.09.011

Kasiński, A. & Ponulak, F. (2006a). Comparison of supervised learning methods for spike time coding in spiking neural networks. *International Journal of Applied Mathematics and Computer Science*, *16*(1), 101–113. doi:10.2417/1200703.0045

Kasiński, A. & Ponulak, F. (2006b). *ReSuMe learning method for Spiking Neural Networks dedicated to neuroprostheses control*. Institute of Control and Information Engineering, Poznan University of Technology. Poznan, Poland. doi:10.1.1.104.7329

Kennedy, P. R. & Bakay, R. A. (1998, June). Restoration of neural output from a paralyzed patient by a direct brain connection. *Neuroreport*, *9*(8), 1707–11.

Kepecs, A., van Rossum, M., Song, S. & Tegner, J. (2002, December). Spike-timing-dependent plasticity: common themes and divergent vistas. *Biological Cybernetics*, *87*(5-6), 446–58. doi:10.1007/s00422-002-0358-6

Kernighan, B. & Pike, R. (1984). *The UNIX Programming Environment* (1st). Englewood Cliffs, NJ, USA: Prentice Hall.

Kerr, A. L., Cheng, S. Y. & Jones, T. A. (2011). Experience-dependent neural plasticity in the adult damaged brain. *Journal of Communication Disorders*, *44*(5), 538–548. doi:10.1016/j.jcomdis.2011.04.011

Khan, M. M., Lester, D., Plana, L. A., Rast, A., Jin, X., Painkras, E. & Furber, S. (2008, June). SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *2008 ieee international joint conference on neural networks (ieee world congress on computational intelligence)* (pp. 2849–2856). Hong Kong, China: IEEE. doi:10.1109/IJCNN.2008.4634199

Kim, K. & Rieke, F. (2001). Temporal Contrast Adaptation in the Input and Output Signals of Salamander Retinal Ganglion Cells. *The Journal of Neuroscience*, *21*(1), 287–299. Retrieved from http://www.jneurosci.org/content/21/1/287.full

Kim, S.-P., Simeral, J. D., Hochberg, L. R., Donoghue, J. P. & Black, M. J. (2008, December). Neural control of computer cursor velocity by decoding motor cortical spiking activity in humans with tetraplegia. *Journal of Neural Engineering*, *5*(4), 455–76. doi:10.1088/1741-2560/5/4/010

Kleim, J. A. & Jones, T. A. (2008). Principles of experience-dependent neural plasticity: implications for rehabilitation after brain damage. *Journal of Speech, Language, and Hearing Research*, *51*(1), S225–39. doi:10.1044/1092-4388(2008/018)

Koessler, L., Maillard, L., Benhadid, A., Vignal, J. P., Felblinger, J., Vespignani, H. & Braun, M. (2009, May). Automated cortical projection of EEG sensors: anatomical correlation via the international 10-10 system. *NeuroImage*, *46*(1), 64–72. doi:10.1016/j.neuroimage.2009.02.006

Kong, K. H., Chua, K. S. G. & Lee, J. (2011). Recovery of upper limb dexterity in patients more than 1 year after stroke: Frequency, clinical correlates and predictors. *NeuroRehabilitation*, *28*(2), 105–111. doi:10.3233/NRE-2011-0639

Konorski, J. (1948). *Conditioned Reflexes and Neuron Organisation.* Cambridge, UK: Cambridge University Press.

Ku, S.-p., Gretton, A., Macke, J. & Logothetis, N. K. (2008, September). Comparison of pattern recognition methods in classifying high-resolution BOLD signals obtained at high magnetic field in monkeys. *Magnetic Resonance Imaging*, *26*(7), 1007–14. doi:10.1016/j.mri.2008.02.016

Kuang, X., Poletti, M., Victor, J. D. & Rucci, M. (2012, March). Temporal Encoding of Spatial Information during Active Visual Fixation. *Current Biology*, *22*(6), 510–514. doi:10.1016/j.cub.2012.01.050

Kudela, P., Franaszczuk, P. J. & Bergey, G. K. (2003, April). Changing excitation and inhibition in simulated neural networks: effects on induced bursting behavior. *Biological Cybernetics*, *88*(4), 276–85. doi:10.1007/s00422-002-0381-7

Kuon, I. & Rose, J. (2006). Measuring the gap between FPGAs and ASICs. In *Proceedings of the 2006 acm-sigda international symposium on field programmable gate arrays - fpga'06* (pp. 21–30). New York, New York, USA: ACM Press. doi:10.1145/1117201.1117205

LaConte, S., Strother, S., Cherkassky, V., Anderson, J. & Hu, X. (2005, June). Support vector machines for temporal classification of block design fMRI data. *NeuroImage*, *26*(2), 317–29. doi:10.1016/j.neuroimage.2005.01.048

Lapicque, L. (1907). Recherches quantitatives sur l'excitation électrique des nerfs traitée comme un polarisation. *Journal de Physiologie et de Pathologie Générale*, *9*, 620–635.

Lauterbur, P. (1973, March). Image Formation by Induced Local Interactions: Examples Employing Nuclear Magnetic Resonance. *Nature*, *242*(5394), 190–191. doi:10.1038/242190a0

LeCun, Y., Kavukcuoglu, K. & Farabet, C. (2010, May). Convolutional networks and applications in vision. In *Proceedings of 2010 ieee international symposium on circuits and systems* (pp. 253–256). Paris, France: IEEE. doi:10.1109/ISCAS.2010.5537907

Lee, H., Kim, Y.-D., Cichocki, A. & Choi, S. (2007, August). Nonnegative tensor factorization for continuous EEG classification. *International Journal of Neural Systems*, *17*(4), 305–17. doi:10.1142/S0129065707001159

Lemm, S., Blankertz, B., Dickhaus, T. & Müller, K.-R. (2011, May). Introduction to machine learning for brain imaging. *NeuroImage*, *56*(2), 387–99. doi:10.1016/j.neuroimage.2010.11.004

Lestienne, R. (1996, January). Determination of the precision of spike timing in the visual cortex of anaesthetised cats. *Biological Cybernetics*, *74*(1), 55–61. doi:10.1007/BF00199137

Leuthardt, E. C., Schalk, G., Wolpaw, J. R., Ojemann, J. G. & Moran, D. W. (2004, June). A brain-computer interface using electrocorticographic signals in humans. *Journal of Neural Engineering*, *1*(2), 63–71. doi:10.1088/1741-2560/1/2/001

Lichtsteiner, P., Posch, C. & Delbruck, T. (2006). A 128 X 128 120db 30mw asynchronous vision sensor that responds to relative intensity change. In *2006 ieee international solid state circuits conference - digest of technical papers* (pp. 2060–2069). San Francisco, CA, USA: IEEE. doi:10.1109/ISSCC.2006.1696265

Lichtsteiner, P., Posch, C. & Delbruck, T. (2008). A 128x128 120 dB 15 μs Latency Asynchronous Temporal Contrast Vision Sensor. *IEEE Journal of Solid-State Circuits*, *43*(2), 566–576. doi:10.1109/JSSC.2007.914337

Lin, C.-Y., Tsai, K.-L., Wang, S.-C., Hsieh, C.-H., Chang, H.-M. & Chiang, A.-S. (2011, March). The Neuron Navigator: Exploring the information pathway through the neural maze. In *2011 ieee pacific visualization symposium* (pp. 35–42). IEEE. doi:10.1109/PACIFICVIS.2011.5742370

Lisman, J. & Spruston, N. (2010, January). Questions about STDP as a General Model of Synaptic Plasticity. *Frontiers in Synaptic Neuroscience*, *2*, 140. doi:10.3389/fnsyn.2010.00140

Litzenberger, M., Kohn, B., Belbachir, A. N., Donath, N., Gritsch, G., Garn, H., … Schraml, S. (2006). Estimation of Vehicle Speed Based on Asynchronous Data from a Silicon Retina Optical Sensor. In *2006 ieee intelligent transportation systems conference* (pp. 653–658). Toronto, Ontario, Canada: IEEE. doi:10.1109/ITSC.2006.1706816

Lodish, H., Berk, A., Kaiser, C. A., Krieger, M., Bretscher, A., Ploegh, H., … Scott, M. P. (2012). *Molecular Cell Biology* (7th). New York, NY, USA: W. H. Freeman.

Lodish, H., Berk, A., Zipursky, S. L., Matsudaira, P., Baltimore, D. & Darnell, J. (2000). *Molecular Cell Biology* (4th). New York, NY, USA: W. H. Freeman.

Logothetis, N. K., Pauls, J., Augath, M., Trinath, T. & Oeltermann, A. (2001, July). Neurophysiological investigation of the basis of the fMRI signal. *Nature*, *412*(6843), 150–7. doi:10.1038/35084005

Lukoševičius, M. (2012). A Practical Guide to Applying Echo State Networks. In G. Montavon, G. B. Orr & K.-R. Müller (Eds.), *Neural networks: tricks of the trade* (2nd, pp. 659–686). Springer Berlin Heidelberg. doi:10.1007/978-3-642-35289-8_36

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

Lyne, A. G., Burgay, M., Kramer, M., Possenti, A., Manchester, R. N., Camilo, F., … Freire, P. C. C. (2004, February). A Double-Pulsar System: A Rare Laboratory for Relativistic Gravity and Plasma Physics. *Science*, *303*(5661), 1153–1157. doi:10.1126/science.1094645

Maass, W. (1997). Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, *10*(9), 1659–1671. doi:10.1016/S0893-6080(97)00011-7

Maass, W. & Markram, H. (2004, December). On the computational power of circuits of spiking neurons. *Journal of Computer and System Sciences*, *69*(4), 593–616. doi:10.1016/j.jcss.2004.04.001

Maass, W., Markram, H. & Natschläger, T. (2002). The "liquid computer": A novel strategy for real-time computing on time series. *Special Issue on Foundations of Information Processing of TELEMATIK*, *8*, 39–43. Retrieved from http://infoscience.epfl.ch/record/117806

Maass, W., Natschläger, T. & Markram, H. (2002, November). Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Computation*, *14*(11), 2531–60. doi:10.1162/089976602760407955

Maneewongvatana, S. & Mount, D. (2001). On the Efficiency of Nearest Neighbor Searching with Data Clustered in Lower Dimensions. In *International conference on computational sciences* (pp. 842–851). Stanford, CA, USA: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=757618

Markram, H., Gerstner, W. & Sjöström, P. J. (2012, January). Spike-timing-dependent plasticity: a comprehensive overview. *Frontiers in Synaptic Neuroscience*, *4*(July), 1–3. doi:10.3389/fnsyn.2012.00002

Markram, H., Lübke, J., Frotscher, M. & Sakmann, B. (1997, January). Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. *Science*, *275*(5297), 213–5. Retrieved from http://www.ncbi.nlm.nih.gov/pubmed/8985014

Marks, S., Estevez, J. E. & Connor, A. M. (2014). Towards the Holodeck. In *Proceedings of the 29th international conference on image and vision computing new zealand* (pp. 42–47). New York, New York, USA: ACM Press. doi:10.1145/2683405.2683424

Marks, S., Estevez, J. E. & Scott, N. M. (2015). Immersive Visualisation of 3-Dimensional Neural Network Structures. In *Proceedings of the 13th international conference on neuro-computing and evolving intelligence*. Auckland, New Zealand: AUT.

Marks, S., Scott, N. M. & Estevez, J. E. (n.d.). Immersive Visualisation of 3-Dimensional Spiking Neural Networks. *Evolving Systems*, (Under Review).

Marnellos, G. & Schreiber, S. (2003). Gene Network Models and Neural Development. In *Modeling neural development* (pp. 27–48). Cambridge, MA, USA: MIT Press.

Marre, O., Yger, P., Davison, A. P. & Frégnac, Y. (2009, November). Reliable recall of spontaneous activity patterns in cortical networks. *The Journal of Neuroscience*, *29*(46), 14596–606. doi:10.1523/JNEUROSCI.0753-09.2009

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press. doi:10.1039/c1ob90002a

Mead, C. (1989). *Analog VLSI and Neural Systems*. Addison-Wesley.

Meier, K., Millner, S., Gr, A., Schemmel, J. & Schwartz, M.-O. (2010). A VLSI Implementation of the Adaptive Exponential Integrate-and-Fire Neuron Model. In *Advances in neural information processing systems* (pp. 1–9). Vancouver, BC, Canada.

Merolla, P., Arthur, J., Alvarez-Icaza, R., Cassidy, A., Sawada, J., Akopyan, F., … Modha, D. S. (2014, August). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, *345*(6197), 668–673. doi:10.1126/science.1254642

Michel, F. C. (1982, January). Theory of Pulsar Magnetospheres. *Reviews of Modern Physics*, *54*(1), 1–66. doi:10.1103/RevModPhys.54.1

Misaki, M., Kim, Y., Bandettini, P. & Kriegeskorte, N. (2010, October). Comparison of multivariate classifiers and response normalizations for pattern-information fMRI. *NeuroImage*, *53*(1), 103–18. doi:10.1016/j.neuroimage.2010.05.051

Mitchell, T., Hutchinson, R., Niculescu, R. S., Pereira, F., Wang, X., Just, M. & Newman, S. (2004). Learning to Decode Cognitive States from Brain Images. *Machine Learning*, *57*(1-2), 145–175.

Mitra, S., Fusi, S. & Indiveri, G. (2009, February). Real-Time Classification of Complex Patterns Using Spike-Based Learning in Neuromorphic VLSI. *IEEE Transactions on Biomedical Circuits and Systems*, *3*(1), 32–42. doi:10.1109/TBCAS.2008.2005781

Mjolsness, E., Sharp, D. H. & Reinitz, J. (1991, October). A connectionist model of development. *Journal of Theoretical Biology*, *152*(4), 429–453. doi:10.1016/S0022-5193(05)80391-1

Mohemmed, A. & Kasabov, N. (2012). Incremental Learning Algorithm for Spatio-Temporal Spike Pattern Classification. In *Ieee world congress on computational intelligence* (pp. 1227–1232). Brisbane, Australia.

Mohemmed, A., Schliebs, S. & Kasabov, N. (2011). SPAN: A neuron for precise-time spike pattern association. In *International conference on neural information processing*. Shanghai, China: Springer.

Mohemmed, A., Schliebs, S., Kasabov, N. & Matsuda, S. (2011). Method for training a spiking neuron to associate input-output spike trains. In *Engineering applications of neural networks* (pp. 219–228). Corfu, Greece: Springer. doi:10.1007/978-3-642-23957-1_25

Mohemmed, A., Schliebs, S., Matsuda, S., Dhoble, K. & Kasabov, N. (2011). SPAN: Spike Pattern Association Neuron for Learning Spatio-Temporal Sequences. *International Journal on Neural Systems*, *22*, 17. doi:10.1142/S0129065712500128

Mohemmed, A., Schliebs, S., Matsuda, S. & Kasabov, N. (2013, May). Training spiking neural networks to associate spatio-temporal input–output spike patterns. *Neurocomputing*, *107*, 3–10. doi:10.1016/j.neucom.2012.08.034

Moradi, S. & Indiveri, G. (2013). An Event-Based Neural Network Architecture With an Asynchronous Programmable Synaptic Memory. In *Ieee transactions on biomedical circuits and systems*. doi:10.1109/TBCAS.2013.2255873

Mori, S. & Zhang, J. (2006, September). Principles of Diffusion Tensor Imaging and Its Applications to Basic Neuroscience Research. *Neuron*, *51*(5), 527–539. doi:10.1016/j.neuron.2006.08.012

Müller, K.-R., Tangermann, M., Dornhege, G., Krauledat, M., Curio, G. & Blankertz, B. (2008, January). Machine learning for real-time single-trial EEG-analysis: from brain-computer interfacing to mental state monitoring. *Journal of Neuroscience methods*, *167*(1), 82–90. doi:10.1016/j.jneumeth.2007.09.022

Naselaris, T., Prenger, R. J., Kay, K. N., Oliver, M. & Gallant, J. L. (2009, September). Bayesian reconstruction of natural images from human brain activity. *Neuron*, *63*(6), 902–15. doi:10.1016/j.neuron.2009.09.006

Naud, R., Marcille, N., Clopath, C. & Gerstner, W. (2008, November). Firing patterns in the adaptive exponential integrate-and-fire model. *Biological Cybernetics*, *99*(4-5), 335–47. doi:10.1007/s00422-008-0264-7

Nawrot, M. P., Schnepel, P., Aertsen, A. & Boucsein, C. (2009, January). Precisely timed signal transmission in neocortical networks with reliable intermediate-range projections. *Frontiers in Neural Circuits*, *3*, 1. doi:10.3389/neuro.04.001.2009

Niedermeyer, E. & Da Silva, F. (2005). *Electroencephalography: Basic Principles, Clinical Applications, and Related Fields*. Phildelphia, PA, USA: Lippincott Williams & Wilkins.

Nita, G. M. & Gary, D. E. (2010, June). The generalized spectral kurtosis estimator. *Monthly Notices of the Royal Astronomical Society: Letters*, no–no. doi:10.1111/j.1745-3933.2010.00882.x

Nuntalid, N., Dhoble, K. & Kasabov, N. (2011). EEG classification with BSA spike encoding algorithm and evolving probabilistic spiking neural network. In *Proceedings of the international conference on neural information processing* (pp. 451–460). Shanghai, China: Springer Berlin Heidelberg. doi:10.1007/978-3-642-24955-6_54

Oesch, N. W. & Diamond, J. S. (2011, October). Ribbon synapses compute temporal contrast and encode luminance in retinal rod bipolar cells. *Nature Neuroscience*, *14*(12), 1555–1561. doi:10.1038/nn.2945

Olsson, J. A. M. & Häfliger, P. (2008). Mismatch reduction with relative reset in integrate-and-fire photo-pixel array. In *2008 ieee biomedical circuits and systems conference* (pp. 277–280). Baltimore, MD, USA: IEEE. doi:10.1109/BIOCAS.2008.4696928

Oostenveld, R. & Praamstra, P. (2001, April). The five percent electrode system for high-resolution EEG and ERP measurements. *Clinical Neurophysiology*, *112*(4), 713–719. doi:10.1016/S1388-2457(00)00527-7

Ozturk, M. C., Xu, D. & Príncipe, J. C. (2007, January). Analysis and Design of Echo State Networks. *Neural Computation*, *19*(1), 111–138. doi:10.1162/neco.2007.19.1.111

Page, S. J., Levine, P., Sisto, S. & Johnston, M. V. (2001). A randomized efficacy and feasibility study of imagery in acute stroke. *Clinical Rehabilitation*, *15*(3), 233–240. doi:10.1191/026921501672063235

Pawlak, V., Wickens, J. R., Kirkwood, A. & Kerr, J. N. D. (2010, January). Timing is not Everything: Neuromodulation Opens the STDP Gate. *Frontiers in Synaptic Neuroscience*, *2*, 146. doi:10.3389/fnsyn.2010.00146

Pearson, M., Gilhespy, I., Gurney, K., Melhuish, C., Mitchinson, B., Nibouche, M. & Pipe, A. (2005). A Real-Time, FPGA Based, Biologically Plausible Neural Network Processor. In W. Duch, J. Kacprzyk, E. Oja & S. Zadrozny (Eds.), *Artificial neural networks: formal models and their applications – {icann 2005}* (Vol. 3697, pp. 755–756). Lecture Notes in Computer Science. Springer Berlin / Heidelberg. doi:10.1007/11550907_161

Perrin, D. (2011, January). Complexity and high-end computing in biology and medicine. *Advances in Experimental Medicine and Biology*, *696*, 377–84. doi:10.1007/978-1-4419-7046-6_38

Pfeil, T., Potjans, T. C., Schrader, S., Potjans, W., Schemmel, J., Diesmann, M. & Meier, K. (2012, January). Is a 4-bit synaptic weight resolution enough? - constraints on enabling spike-timing dependent plasticity in neuromorphic hardware. *Frontiers in Neuroscience*, *6*, 90. doi:10.3389/fnins.2012.00090. arXiv: arXiv:1201.6255v5

Pohlmeyer, E., Solla, S., Perreault, E. & Miller, L. (2007, December). Prediction of upper limb muscle activity from motor cortical discharge during reaching. *Journal of Neural Engineering*, *4*(4), 369–79. doi:10.1088/1741-2560/4/4/003

Ponulak, F. (2005). *ReSuMe - New supervised learning method for Spiking Neural Networks*. Institute of Control and Information Engineering, Poznan University of Technology. Poznan, Poland: Citeseer. doi:10.1.1.60.6325

Ponulak, F. & Kasiński, A. (2006). Generalization Properties of Spiking Neurons trained with ReSuMe method. In *14th european symposium on artificial neural networks* (pp. 629–634). Bruges, Belgium.

Popović, D. B. & Sinkjaer, T. (2000). *Control of Movement for the Physically Disabled*. London: Springer. doi:10.1007/978-1-4471-0433-9

Qiao, N., Mostafa, H., Corradi, F., Osswald, M., Stefanini, F., Sumislawska, D. & Indiveri, G. (2015, April). A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses. *Frontiers in Neuroscience*, *9*. doi:10.3389/fnins.2015.00141

Quian Quiroga, R. & Panzeri, S. (2009). Extracting information from neuronal populations: information theory and decoding approaches. *Nature Reviews Neuroscience*, *10*(3), 173–185. doi:10.1038/nrn2578

Ramakrishnan, S., Hasler, P. & Gordon, C. (2010, May). Floating gate synapses with spike time dependent plasticity. In *Proceedings of the ieee international symposium on circuits and systems* (pp. 369–372). Paris, France: IEEE. doi:10.1109/ISCAS.2010.5537768

Rast, A., Galluppi, F., Davies, S., Plana, L. A., Patterson, C., Sharp, T., … Furber, S. (2011, November). Concurrent heterogeneous neural model simulation on real-time neuromimetic hardware. *Neural Networks*, *24*(9), 961–978. doi:10.1016/j.neunet.2011.06.014

Rast, A., Jin, X., Galluppi, F., Plana, L. A., Patterson, C. & Furber, S. (2010). Scalable event-driven native parallel processing. In *Proceedings of the 7th acm international conference on computing frontiers* (p. 21). New York, New York, USA: ACM Press. doi:10.1145/1787275.1787279

Rast, A., Stokes, A. B., Rowley, A. G., Davies, S., Lester, D., Furber, S., … Djurfeldt, M. (2015). AERIE-P: AER Intersystem Exchange Protocol. Capo Caccia, Sardinia, Italy: Universal AER Communications over Ethernet for Multi-System Applications Workgroup, 2015 CSN Cognitive Neuromorphic Engineering Workshop.

Raymond, E. (2003). *The Art of Unix Programming* (1st). Addison-Wesley.

Rieke, F. (2001). Temporal Contrast Adaptation in Salamander Bipolar Cells. *The Journal of Neuroscience*, *21*(23), 9445–9454. Retrieved from http://www.jneurosci.org/content/21/23/9445.full

Rizzolatti, G., Fogassi, L. & Gallese, V. (2001). Neurophysiological mechanisms underlying the understanding and imitation of action. *Nature reviews. Neuroscience*, *2*(9), 661–670. doi:10.1038/35090060

Rolls, E. T. & Treves, A. (2011). The neuronal encoding of information in the brain. *Progress in Neurobiology*, *95*(3), 448–490. doi:10.1016/j.pneurobio.2011.08.002

Sastre-Garriga, J., Galán-Cartañá, I., Montalbán, X. & Thompson, A. (2005, June). Neurorehabilitation in multiple sclerosis. *Neurologia*, *20*(5), 245–54. Retrieved from http://www.ncbi.nlm.nih.gov/pubmed/15954034

Schiller, U. D. & Steil, J. J. (2005, January). Analyzing the weight dynamics of recurrent learning algorithms. *Neurocomputing*, *63*, 5–23. doi:10.1016/j.neucom.2004.04.006

Schliebs, S. (2010). *Heterogeneous Probabilistic Models for Optimisation and Modelling of Evolving Spiking Neural Networks* (Doctoral Thesis, Auckland University of Technology).

Schliebs, S., Capecci, E. & Kasabov, N. (2013). Spiking Neural Network for On-line Cognitive Activity Classification Based on EEG Data. In *Proceedings of the international conference on neural information processing* (pp. 55–62). Daegu, Korea. doi:10.1007/978-3-642-42051-1_8

Schliebs, S. & Kasabov, N. (2013, June). Evolving spiking neural network - a survey. *Evolving Systems*, *4*(2), 87–98. doi:10.1007/s12530-013-9074-9

Schliebs, S., Nuntalid, N. & Kasabov, N. (2010). Towards spatio-temporal pattern recognition using evolving spiking neural networks. In *Proceedings of the international conference on neural information processing* (pp. 163–170). Sydney, Australia: Springer. doi:10.1007/978-3-642-17537-4_21

Schrauwen, B. & van Campenhout, I. (2003). BSA, a fast and accurate spike train encoding scheme. In *Proceedings of the international joint conference on neural networks* (pp. 2825–2830). Portland, Oregon, US: IEEE. doi:10.1109/IJCNN.2003.1224019

Schulz, J. M. (2010, January). Synaptic Plasticity in vivo: More than Just Spike-Timing? *Frontiers in Synaptic Neuroscience*, *2*, 150. doi:10.3389/fnsyn.2010.00150

Scott, N. M. (2012). *A Study of the Behaviour of Incremental 'Spike Pattern Association Neuron' Learning Algorithm for Spiking Neural Networks* (Honours Dissertation, Auckland University of Technology).

Scott, N. M. & Kasabov, N. (2015). Feasibility of Implementing NeuCube on the SpiNNaker Neuromorphic Hardware Device. In *Proceedings of the 13th international conference on neuro-computing and evolving intelligence*. Auckland, New Zealand: AUT.

Scott, N. M., Kasabov, N. & Indiveri, G. (2013). NeuCube Neuromorphic Framework for Spatio-Temporal Brain Data and Its Python Implementation. In *Proceedings of the international conference on neural information processing* (pp. 78–84). Daegu, Korea: Springer. doi:10.1007/978-3-642-42051-1_11

Scott, N. M., Mahmoud, M., Hartono, R. N., Gulyaev, S. & Kasabov, N. (2015). Feasibility analysis of using the NeuCube Spiking Neural Network Architecture for Dispersed Transients and Pulsar Detection. In *Proceedings of the 13th international conference on neuro-computing and evolving intelligence*. Auckland, New Zealand: AUT.

Sengupta, N., Scott, N. M. & Kasabov, N. (2015). Framework For Knowledge Driven Data Encoding For Brain Data Modelling Using Spiking Neural Network Architecture. In *Proceedings of the 5th international conference on fuzzy and neural computing*. Hyderabad, India: Springer.

Serafino, N. & Zaghloul, M. (2013, August). Review of nanoscale memristor devices as synapses in neuromorphic systems. In *2013 ieee 56th international midwest symposium on circuits and systems (mwscas)* (pp. 602–603). IEEE. doi:10.1109/MWSCAS.2013.6674720

Serrano-Gotarredona, T., Linares-Barranco, B., Galluppi, F., Plana, L. A. & Furber, S. (2015, May). ConvNets experiments on SpiNNaker. In *2015 ieee international symposium on circuits and systems (iscas)* (pp. 2405–2408). IEEE. doi:10.1109/ISCAS.2015.7169169

Serruya, M. D., Hatsopoulos, N. G., Paninski, L., Fellows, M. R. & Donoghue, J. P. (2002, March). Instant neural control of a movement signal. *Nature*, *416*(6877), 141–2. doi:10.1038/416141a

Sharp, T., Galluppi, F., Rast, A. & Furber, S. (2012, September). Power-efficient simulation of detailed cortical microcircuits on SpiNNaker. *Journal of Neuroscience Methods*, *210*(1), 110–118. doi:10.1016/j.jneumeth.2012.03.001

Sjöström, P. J., Turrigiano, G. G. & Nelson, S. B. (2001, December). Rate, Timing, and Cooperativity Jointly Determine Cortical Synaptic Plasticity. *Neuron*, *32*(6), 1149–1164. doi:10.1016/S0896-6273(01)00542-6

Soltic, S. & Kasabov, N. (2010). Knowledge extraction from evolving spiking neural networks with rank order population coding. *International Journal of Neural Systems*, *20*(6), 437–445.

Song, Q. & Kasabov, N. (2001). EC:, A Novel On-line, Evolving Clustering Method and its Applications. In *Proceedings of the fifth biannual conference on artificial neural networks and expert systems* (pp. 97–92). Dunedin, New Zealand.

Song, S., Miller, K. D. & Abbott, L. F. (2000, September). Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neuroscience*, *3*(9), 919–26. doi:10.1038/78829

Stam, C. J. (2004, January). Functional connectivity patterns of human magnetoencephalographic recordings: a 'small-world' network? *Neuroscience Letters*, *355*(1-2), 25–8.

Stefanini, F., Neftci, E. O., Sheik, S. & Indiveri, G. (2014, August). PyNCS: a microkernel for high-level definition and configuration of neuromorphic electronic systems. *Frontiers in Neuroinformatics*, *8*(73). doi:10.3389/fninf.2014.00073

Stein, R. B., Gossen, E. R. & Jones, K. E. (2005, May). Neuronal variability: noise or part of the signal? *Nature Neuroscience*, *6*(5), 389–97. doi:10.1038/nrn1668

Storjohann, R. & Marcus, G. F. (2005). NeuroGene: integrated simulation of gene regulation, neural activity and neurodevelopment. In *Proceedings of the ieee international joint conference on neural networks* (Vol. 2, pp. 428–433). Montreal, Quebec, Canada: IEEE. doi:10.1109/IJCNN.2005.1555869

Storlie, C., Sexton, J., Pakin, S., Lang, M., Reich, B. & Rust, W. (2014, December). Modeling and Predicting Power Consumption of High Performance Computing Jobs. arXiv: 1412.5247

Strukov, D. B., Snider, G. S., Stewart, D. R. & Williams, R. S. (2008, May). The missing memristor found. *Nature*, *453*(7191), 80–83. doi:10.1038/nature06932

Svirskis, G. & Rinzel, J. (2000, August). Influence of temporal correlation of synaptic input on the rate and variability of firing in neurons. *Biophysical Journal*, *79*(2), 629–37. doi:10.1016/S0006-3495(00)76321-1

Talairach, J. & Tournoux, P. (1988). *Co-planar Stereotaxic Atlas of the Human Brain: 3-Dimensional Proportional System*. New York, NY, USA: Theirne Medical Publishers.

Taylor, D. M., Tillery, S. I. H. & Schwartz, A. B. (2002, June). Direct cortical control of 3D neuroprosthetic devices. *Science*, *296*(5574), 1829–32. doi:10.1126/science.1070291

Taylor, D., Chamberlain, J., Signal, N., Scott, N. M., Kasabov, N., Capecci, E., … Hou, Z.-G. (2015). Brain-Computer Interfaces for Neuro Rehabilitation. In *Proceedings of the 13th international conference on neuro-computing and evolving intelligence*. Auckland, New Zealand: AUT.

Taylor, D., Scott, N. M., Kasabov, N., Capecci, E., Tu, E., Saywell, N., … Hou, Z.-G. (2014, July). Feasibility of NeuCube SNN architecture for detecting motor execution and motor intention for use in BCI applications. In *2014 international joint conference on neural networks (ijcnn)* (pp. 3221–3225). IEEE. doi:10.1109/IJCNN.2014.6889936

Taylor, J. H. (1991, July). Millisecond pulsars: nature's most stable clocks. *Proceedings of the IEEE*, *79*(7), 1054–1062. doi:10.1109/5.84982

Taylor, J. H. & Weisberg, J. M. (1989, October). Further experimental tests of relativistic gravity using the binary pulsar PSR 1913 + 16. *The Astrophysical Journal*, *345*, 434. doi:10.1086/167917

Thompson, S. E. & Parthasarathy, S. (2006). Moore's Law: the Future of SI Microelectrics. *Materials Today*, *9*(6), 20–25.

Thorpe, S. (2001). Spike-based strategies for rapid processing. *Neural Networks*, *14*(6-7), 715–25.

Thorpe, S., Fize, D. & Marlot, C. (1996, June). Speed of processing in the human visual system. *Nature*, *381*(6582), 520–2. doi:10.1038/381520a0

Thorpe, S. & Gautrais, J. (1998). Rank order coding. In *Proceedings of the 6th annual conference on computational neuroscience* (pp. 113–118). New York, NY, USA: Plenum Press.

Thorpe, S., Guyonneau, R., Guilbaud, N., Allegraud, J.-M. & van Rullen, R. (2004, June). SpikeNet: real-time visual processing with one spike per neuron. *Neurocomputing*, *58-60*, 857–864. doi:10.1016/j.neucom.2004.01.138

Thulborn, K. R., Waterton, J. C., Matthews, P. M. & Radda, G. K. (1982, February). Oxygenation dependence of the transverse relaxation time of water protons in whole blood at high field. *Biochimica et Biophysica Acta (BBA) - General Subjects*, *714*(2), 265–270. doi:10.1016/0304-4165(82)90333-6

Tomioka, R. & Dornhege, G. (2006). *Spectrally weighted common spatial pattern algorithm for single trial EEG classification* (tech. rep. No. July). Department of Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo. Tokyo, Japan. Retrieved from http://www.ibis.t.u-tokyo.ac.jp/ryotat/metrCSP-SPEC.pdf

Tuckwell, H. C. (1988). Nonlinear and Stochastic Theories. In *Introduction to theoretical neurobiology*. Cambridge University Press.

van Rossum, M. (2001, April). A Novel Spike Distance. *Neural Computation*, *13*(4), 751–763. doi:10.1162/089976601300014321

Venkateswara, K. R., Govardhan, A. & Chalapati, K. V. R. (2012, February). Spatiotemporal Data Mining: Issues, Tasks And Applications. *International Journal of Computer Science & Engineering Survey*, *3*(1), 39–52. doi:10.5121/ijcses.2012.3104

Verstraeten, D., Schrauwen, B., D'Haene, M. & Stroobandt, D. (2007). An experimental unification of reservoir computing methods. *Neural Networks*, *20*(3), 391–403.

Victor, J. D. & Purpura, K. (1997). Metric-space analysis of spike trains: Theory, algorithms, and application. *Network: Computation in Neural Systems*, *8*(2), 127–164.

Villa, A. E., Tetko, I. V., Hyland, B. & Najem, A. (1999, February). Spatiotemporal activity patterns of rat cortical neurons predict responses in a conditioned task. *Proceedings of the National Academy of Sciences of the United States of America*, *96*(3), 1106–11.

Vogels, T. P. & Abbott, L. F. (2005, November). Signal propagation and logic gating in networks of integrate-and-fire neurons. *The Journal of Neuroscience*, *25*(46), 10786–95. doi:10.1523/JNEUROSCI.3508-05.2005

Vogelstein, R. J., Mallik, U., Culurciello, E., Cauwenberghs, G. & Etienne-Cummings, R. (2007, September). A multichip neuromorphic system for spike-based visual information processing. *Neural Computation*, *19*(9), 2281–300. doi:10.1162/neco.2007.19.9.2281

von Kapri, A., Rick, T., Potjans, T. C., Diesmann, M. & Kuhlen, T. (2011). Towards the visualization of spiking neurons in virtual reality. *Studies in Health Technology and Informatics*, *163*, 685–7.

Watts, M. (2009, May). A Decade of Kasabov's Evolving Connectionist Systems: A Review. *IEEE Transactions on Systems, Man, and Cybernetics*, *39*(3), 253–269. doi:10.1109/TSMCC.2008.2012254

Watts, M. & Kasabov, N. (1999). Neuro-Genetic Information Processing for Optimisation and Adaptation in Intelligent Systems. In *Studies in fuzziness and soft computing* (Chap. 6, Vol. 30, pp. 97–110).

Wendling, F., Bartolomei, F., Bellanger, J. J. & Chauvel, P. (2002, May). Epileptic fast activity can be explained by a model of impaired GABAergic dendritic inhibition. *The European Journal of Neuroscience*, *15*(9), 1499–508.

Wysoski, S. G., Benuskova, L. & Kasabov, N. (2010). Evolving spiking neural networks for audiovisual information processing. *Neural Networks*, *23*(7), 819–35.

Yang, M., Liu, S.-C. & Delbruck, T. (2015, September). A Dynamic Vision Sensor With 1% Temporal Contrast Sensitivity and In-Pixel Asynchronous Delta Modulator for Event Encoding. *IEEE Journal of Solid-State Circuits*, *50*(9), 2149–2160. doi:10.1109/JSSC.2015.2425886

Yu, T. & Cauwenberghs, G. (2010, June). Analog VLSI Biophysical Neurons and Synapses With Programmable Membrane Channel Kinetics. *IEEE Transactions on Biomedical Circuits and Systems*, *4*(3), 139–148. doi:10.1109/TBCAS.2010.2048566

Zamarreño-Ramos, C., Camuñas-Mesa, L. A., Pérez-Carrasco, J. A., Masquelier, T., Serrano-Gotarredona, T. & Linares-Barranco, B. (2011). On Spike-Timing-Dependent-Plasticity, Memristive Devices, and Building a Self-Learning Visual Cortex. *Frontiers in Neuroscience*, *5*. doi:10.3389/fnins.2011.00026

Zhang, C. & Budgen, D. (2012). What do we know about the effectiveness of software design patterns? *IEEE Transactions on Software Engineering*, *38*(5), 1–19. doi:10.1109/TSE.2011.79

Zimmermann, R., Marchal-Crespo, L., Edelmann, J., Lambercy, O., Fluet, M.-C., Riener, R., … Gassert, R. (2013). Detection of motor execution using a hybrid fNIRS-biosignal BCI: a feasibility study. *Journal of Neuroengineering and Rehabilitation*, *10*(1), 4. doi:10.1186/1743-0003-10-4

Abbreviations

AdEx Adaptive Exponential Integrate-And-Fire . 35, 36, 86, 90, 149, 150, 157, 158, 161, 162, 167

AER Address-Event Representation 39, 40, 103, 116–118, 141, 152, 153, 158, 159, 162, 174, 206

AMPA α-Amino-3-Hydroxy-5-Methyl-4-Isoxazolepropionic Acid . . . 62

ANN Artificial Neural Network 21

ASIC Application-Specific Integrated Circuit 48, 86, 90, 97, 104, 117, 118, 144, 145, 147–149, 151, 156, 159, 160, 163, 164, 167, 198, 199, 202, 207

BCI Brain Computer Interface 1, 4, 9, 11, 69, 112, 149, 154, 161, 162, 178, 202, 203, 205, 240, 241, 244, 246–248

BOLD Blood Oxygenation Level Dependent 13, 14

BSA Ben's Spiker Algorithm 11, 38, 40–42, 72, 86, 87

CMOS Complementary Metal-Oxide Semiconductor 153, 157

CNGM Computational Neuro-Genetic Model 25, 48, 60, 61, 75, 94–96, 98, 99, 111, 127, 139, 177, 246

CSV Comma-Separated Value 103, 109, 126

deSNN Dynamic Evolving Spiking Neural Network Classifier 22, 32, 45, 56, 58, 59, 72, 75, 94, 107, 114, 115, 125, 158, 175, 243, 245, 253

DTI Diffusion Tensor Imaging . . 12, 15, 74, 92, 93, 111–113, 238, 247

DVS Dynamic Vision Sensor 39, 116, 117, 152, 153, 168, 173

eCoS Evolving Connectionist System 56, 57

EEG Electroencephalography 1, 9–12, 19, 22, 38, 42, 43, 87, 88, 91, 116, 132, 188, 203, 235–238, 240–242, 244, 245, 247, 248

ETH Zürich Eidgenössische Technische Hochschule Zürich 156

ESN Echo State Network 62–64

eSNN Evolving Spiking Neural Network 56–59

fMRI 'Functional' Magnetic Resonance Imaging 12–14, 19, 43, 87, 89, 90, 92, 203, 236–238, 247

FPGA Field Programmable Gate Array . 144, 147–149, 155, 156, 161, 163, 173, 174, 191, 195, 196, 198, 202, 207

GA Genetic Algorithm . . . 44
GABA γ-Aminobutyric Acid . . . 25, 62, 95
GRN Gene Regulatory Network . . . 60–62, 95

HDL Hardware Description Language . . . 147, 157, 159
HH Hodgkin-Huxely . . . 26, 89
HMD Head Mounted Display . . . 78–82
HMM Hidden Markov Model . . . 240

INI Institute for Neuroinformatics 117, 118, 153, 156, 159, 167, 202, 207
IO Input-Output . 103, 110, 111, 116, 118, 122, 148, 168, 170, 173, 174, 178, 184–186, 188, 191, 192, 198, 207

JSON JavaScript Object Notation . . . 103, 116, 118, 119, 155, 187

***k*NN** *k*-Nearest-Neighbour . . . 14, 132

LIF Leaky Integrate-And-Fire 30, 32, 33, 35, 74, 89, 105, 167, 168, 176, 196, 243, 253
LSM Liquid State Machine . . . 63, 64, 70, 73, 74, 89, 99, 137
LTD Long-Term Depression . . . 46, 49, 175
LTP Long-Term Potentiation . . . 46, 49, 175
LUT Lookup Table . . . 48

MI Motor Imagery . . . 239–241
MLP Multi-Layer Perceptron . . . 1, 19, 21, 107, 235
MNI Montreal Neurological Institute . . . 91, 93, 112, 191, 236, 247
MNP Motor Neuron Prosthetic . . . 112, 154
MPI Message-Passing Interface . . . 67
MRI Magnetic Resonance Imaging . . . 12, 13, 15, 88

NMDA N-Methyl-D-Aspartic Acid . . . 62, 158

pLIF Probabilistic Leaky Integrate-And-Fire 35, 74, 86, 89, 90, 176, 177
PSP Post-Synaptic Potential . . . 61, 62

ReSuMe Remote Supervised Method . . . 45, 50, 53, 54
RFI Radio Frequency Interference . . . 18
RMSE Root Mean Squared Error . . . 43
RNG Random Number Generation . . . 90, 176
RNN Recurrent Neural Network . . . 19, 62–64, 70
RO Rank-Order Learning . . . 57–59
ROLLS Reconfigurable On-Line Learning Spiking 156–158, 160, 199, 202, 207

SDSP Spike Dependent Synaptic Plasticity 45–47, 49, 50, 58, 59, 151, 155, 158
SKA Square Kilometre Array 17, 18, 204, 254, 255
SNN Spiking Neural Network 2, 4, 11, 14, 20, 21, 24, 25, 28, 37, 44–46, 50, 57, 63, 66, 68, 70, 76, 83, 85, 90, 94–96, 98, 99, 101, 107, 108, 121, 135, 136, 143, 162, 164, 167, 179, 182, 184, 192, 197, 198, 200, 201, 203–207, 235, 245, 247, 255
SNR Signal-to-Noise Ratio 40, 252–254
SPAN Spike Pattern Association Neuron 22, 45, 46, 50, 51, 53–56, 75, 94, 161
SRM Spike Response Model 30
SSTD Spatio- and Spectro-Temporal Data 1–3, 6–9, 15, 18–20, 22, 69–72, 74, 76, 83, 98, 138, 204, 205, 209
STDP Spike Time Dependent Plasticity . 45–51, 58, 59, 74, 123, 133, 134, 137, 151, 155, 156, 165, 168, 174, 175, 186, 198, 253
STFT Short-Term Fourier Transform 18, 19
SVM Support Vector Machine . . . 1, 12, 14, 19, 63, 75, 94, 107, 114, 235

TC Threshold-Based Temporal Contrast 39
TD Threshold-Based Temporal Difference 38, 39, 87, 99, 116, 121, 206, 243, 252, 253

VLSI Very Large Scale Integration 48, 86, 90, 97, 117, 140, 145, 148–151, 153, 156, 197, 199, 202, 207

Appendix A

Case Study in Neuroinformatics

The NeuCube framework was initially introduced for the classification and interpretation of neuroinformatics data – that is, information about the brain and its dynamics. Neuroinformatics data is a difficult challenge in machine learning, primarily due to its inherent spatial and temporal dynamics. Classical machine learning algorithms (*e.g.* SVM or MLP) also struggle to effectively represent its non-linearity and non-stationary nature.

The NeuCube explicitly incorporates these aspects through the use of a meaningfully-structured SNN reservoir. This reservoir captures the spatial links within the data through its 3D representation of neurons, along with a representation of the data's temporal dynamics through those neurons' spiking behaviour.

In Section 2.1.1, some of the most common sources of neuroinformatics data were discussed. Here, we will introduce how we can use these to develop effective NeuCube architectures. Additionally, we establish a case study on the classification of brain signals generating complex natural hand movements, captured through EEG. This case study provides evidence that the NeuCube can adequately deal with complex spatio-temporal data.

A.1 NeuCube Architectures for Neuroinformatics

As the nature and structure of the brain is relatively well known – at least, with regards to the data collection devices we have discussed – it is entirely possible for us to generate meaningful reservoir structures which capture both the spatial, and the temporal, dynamics of the data. Please, refer to Section 2.1.1 for a review of these technologies.

Here, we briefly discuss the basic architectures for the design of NeuCube reservoirs for EEG and fMRI, our most significant neuroinformatics data sources. These are the starting point for the generation of our actual reservoir architecture; they should be read in the context of the design methodology given in Chapter 5.

A.1.1 Electroencephalography

When structuring a reservoir to represent EEG data, we have some *a-priori* information about the nature and source of the data. It is, clearly, generated by the brain, and recorded in known locations on the scalp. The issue then, is to reconcile these two sources of location information – the brain structure and the electrode locations – with each other into one cohesive structure.

Recall that EEG can be associated with a known brain atlas. In this case, we have either the Talairach or MNI systems, which each represent the brain as a cloud of 3D points. These two systems are largely interchangeable; the Talairach altlas is more commonly used, but the MNI system is quickly supplanting it as it is more accurate. In any case, we can associate an EEG collection location with a known brain location. Effectively, we take a selection of points in one of these spaces, and associate the input locations with the neuron in the reservoir that as closely as possible represents their real location.

The reservoir, then, is structured using our knowledge of the Talairach or MNI atlases. The network size is heuristically chosen, based on the complexity of the data; more complex or longer data will require a larger network, as memory and computational capacity of such reservoirs is closely correlated with network size. See Section 5.2 for further details on the selection of reservoir parameters.

To provide the locations of the EEG electrodes, the EEG to cortical location mapping given in Koessler et al. (2009) (*cf.* Table A.1) has been used. This is a comprehensive listing of 10-10 system electrode locations in Talairach space, generated through colocation of EEG electrodes with fMRI. Due to this, we can be sure that the input locations given are representative of the actual cortical locations of the electrodes.

Therefore, from this, it is a trivial matter to reconcile these two sets of points. In order to implement this effectively, we utilise the technique given in Section 6.4.3.3.

To date, of the papers published applying the NeuCube to EEG data all have used some variation of this mapping. These papers include Capecci et al. (2015), Chen

10-10	Talairach Coordinates			Gyri		BA
Label	x	y	z			
F7	-52.1 ± 3.0	28.6 ± 6.4	3.8 ± 5.6	L FL	Inferior Frontal G	45
T7	-65.8 ± 3.3	17.8 ± 6.8	-2.9 ± 6.1	L TL	Middle Temporal G	21
PO4	35.2 ± 6.4	-82.6 ± 6.4	26.1 ± 9.7	R OL	Superior Occipital G	19
⋮						
Cz	0.8 ± 4.9	-21.9 ± 9.4	77.4 ± 6.7	M FL	Precentral G	4
O2	25.0 ± 5.7	-95.2 ± 5.8	6.2 ± 11.4	R OL	Middle Occipital G	18

Table A.1: Example locations of EEG collection positions in the International 10-10 System mapped to cortical positions in Talairach space (Koessler et al., 2009). These locations are used to select the closest neuron in the reservoir to act as an input location for the data from that channel. 'BA' represents the Broadmann Area number.

et al. (2013), J. Hu et al. (2014), Kasabov and Capecci (2015), Kasabov, Hu et al. (2013), Kasabov et al. (2015), Scott et al. (2013) and D. Taylor et al. (2014).

This mapping and network structure were introduced in Scott et al. (2013). Simultaneously, it was used in Kasabov, Hu et al. (2013), and by Schliebs et al. (2013). This mapping is a contribution of this thesis. The simultaneous publication is due to those three papers being published in the same conference.

A.1.2 Functional Magnetic Resonance Imaging

The generation of a network structure for fMRI is a more straightforward process than for EEG. In effect, as a part of the fMRI itself we have the structure of the network. This is given in the positions of the voxels. Each voxel is associated with a 3D location in the brain, and also contains the temporal information of the data. It is therefore a trivial matter to structure a network using these voxel locations to represent both the reservoir neurons, and the input locations.

The only issue with this technique is that of scale. A high-resolution fMRI scan can contain upwards of 100,000 voxels per time slice. Unfortunately it is not always feasible – or, indeed, even *possible* in the case of some neuromorphic systems – to implement this scale of network. To resolve this, we must perform some dimensionality reduction. This is chiefly implemented using simple neighbourhood averaging. In essence, we average some neighbourhood of voxels togther into one super-voxel. This reduces the dimensionality of the data, but along with it, the resolution. A more

sophisticated technique such as Principle Component Analysis or similar may be equally appropriate here, but as we are trying to keep as much of the data intact as possible, we should try to avoid this type of dimensionality reduction. This decision must be made heuristically, in the context of the design methodology discussed in Chapter 5. In fact, a much better approach is to select a computational platform which can handle the dimensionality. Issues such as this support the development of the neuromorphic systems version of the NeuCube introduced in Chapter 7.

An example of this base architecture applied to fMRI is given in the recent papers of Doborjeh et al. (2014a) and Doborjeh et al. (2014b), where this concept was extended and applied to the classification and segmentation of resting state and language task data.

A.1.3 Diffusion Tensor Imaging

Diffusion Tensor Imaging (DTI) is not an appropriate source of information for reservoir structure in and of itself; instead, it must be paired with some temporal source of brain data such as those discussed above. Why, then, is DTI useful in this context?

DTI is useful in that it can assist us in generating an informed design for the general *connectome* of a reservoir. Instead of using the small-world connectivity discussed in Section 6.4.3.1, we can instead use an approximation of the actual subject's real connectome in place. This, along with a meaningfully shaped reservoir, is intended to retain the spatio-temporal links within the data with more accuracy than a general connectome.

While this concept has yet to be applied in an empirical study, it is supported by the general trend exhibited by these architectures. That is, the more *a-priori* information we have about the nature and source of the data, the better the reservoir can represent it in both spatial and temporal dynamics.

A.2 Classification of Complex Natural Hand Movements utilising Electroencephalography

We performed an experiment to gauge the NeuCube's ability to classify complex neuroinformatics data, in an attempt to provide empirical evidence to the claims in this thesis. Here, an experiment that established the NeuCube was effective at classifying complex natural hand movements as captured through EEG data is

introduced. Additionally, we established that in this case, the NeuCube framework was more effective than a number of contemporary machine learning techniques when applied to the same task.

N.B.: This study was originally published as D. Taylor et al. (2014) and discussed further in D. Taylor et al. (2015). It has been adapted from there, but is largely as it was originally published.

A.2.1 Motivation & Research Questions

A focal neurological insult that causes changes to cerebral blood flow, such as in a stroke, can result in mild to severe motor dysfunctions on the contralateral side of the body. Although some spontaneous recovery usually occurs in the first 6 months after stroke only about 14% of people with stroke recover normal use of the upper limb (Kong, Chua & Lee, 2011). The driver of functional recovery after stroke is neural plasticity, the propensity of synapses and neuronal circuits to change in response to experience and demand (Fox, 2009; Kerr, Cheng & Jones, 2011; Kleim & Jones, 2008). Whilst it is known that frequency and intensity of intervention following stroke is important high intensity rehabilitation is resource-limited. In order to deliver interventions at a high enough intensity and frequency for neural plasticity we need to develop devices that can assist with rehabilitation without the concentrated input of rehabilitation professionals.

Motor Imagery (MI), or the mental rehearsal of a movement, is an approach used by rehabilitation professionals to encourage motor practice in the absence of sufficient muscle activity (Jeannerod, 1994, 2001; Rizzolatti, Fogassi & Gallese, 2001). MI is thought to activate similar cortical networks as activated in a real movement, including activation of the primary motor cortex, premotor cortex, supplementary motor area and parietal cortices (Fadiga, Fogassi, Pavesi & Rizzolatti, 1995; Grush, 2004). Recent evidence suggests that although there are common cortical networks in real and imagined movement (frontal and parietal sensorimotor cortices) there are also important differences, with ventral areas being activated in imagined movement, but not in real movement. These specific additional activations in the extreme/external capsule may represent an additional cognitive demand of imagery based tasks.

Recovery of movement control is greater after motor execution training than after MI training alone. Interestingly the combination of MI training with even passive movement generates greater recovery than MI alone (Page, Levine, Sisto & Johnston,

2001). Combining motor imagery with functional electrical muscle stimulation, via BCI devices, may result in greater neural plasticity and recovery than motor imagery alone, or motor imagery combined with passive movement. The additional feedback to the brain provided by executing a movement may enhance plasticity and reduce the cognitive demand of motor imagery. Many people following stroke or other neurological disorder have some residual muscle activity but fail to recruit enough motor units at an appropriate speed and pattern, to generate sufficient force to complete the desired movement (Chang, Zhou, Rymer & Li, 2013; Gray, Rice & Garland, 2012). A BCI device in which motor imagery triggers an appropriate signal to a functional electrical stimulation system would facilitate the practice of real movements and potentially result in greater neural plasticity and functional recovery.

EEG records brain signals through electrodes on the scalp and is the most widely used method for recording brain data used in BCI devices. EEG is non-invasive and has good temporal and spatial resolution. However, EEG systems have been criticized because of the time consuming and complex training period for the potential user (Zimmermann et al., 2013). One advantage of the NeuCube framework is that intensive training of the user is not required. This is due to the fact that the NeuCube classifies naturally elicited cortical activity, rather than a specific component of the EEG signal such as the P300 wave, the production of which has to be learned by the user. In addition, the NeuCube is capable of learning in an on-line fashion, training as it is used.

Here, we are investigating the feasibility of using NeuCube with EEG data to develop a functional electrical stimulation BCI system that is able to assist in the rehabilitation of complex upper limb movements. Two methods of use are under consideration, firstly for people who have no voluntary activity in a limb who would drive the device using MI, and secondly for people who have some residual activity in their muscles that, in addition to using MI, may augment the device with their own muscle activity. To do this it is important to establish a high degree of accuracy of classification of movement intention and movement execution to ensure that the appropriate electrical stimulation output is then provided. One of the challenges to any BCI system is the extent to which it accurately classifies the input signal.

In Zimmermann et al. (2013) real movement, consisting of a pinch grip to a specified force level, compared to a resting state, was used. Data were collected using functional Near Infrared Spectrometry (fNIRS) combined with other physiological data, such as blood pressure and respiratory information. Using a Hidden Markov Model (HMM)

as the classifier framework accuracies ranging between 79.6% and 98.8% over 2 classes were achieved. Using fNIRS in a trial of MI, Zimmermann et al. (2013) investigated the classification accuracy of a simple imagined tap of the thumb on a keyboard versus a complex multi-digit tapping sequence. Linear discriminant analysis (LDA) was used in combination with careful selection of the best performing data channel (out of 3 possible channels) and best 4 features for each participant. The study in Holper and Wolf (2011) reported classification accuracies in a 2-class model (simple imagined movement or complex imagined movement) of between 70.8% and 91.7%. A Sparse Common Spatial Pattern optimization technique that reduced EEG channels by disregarding noisy channels and channels thought to be irrelevant was reported in Arvaneh, Guan, Ang and Quek (2011); however, this approach results in a loss of data that could be informative.

We were interested in determining if it was feasible to use the NeuCube framework as a driver of BCI devices. As a first step we wanted to determine if the NeuCube was at least equivalent in classifying movement tasks as other commonly used methods. As proof-of concept we designed a study that required NeuCube to classify imagined and real movements in two different directions and at rest (wrist flexion, extension or rest). The general hypothesis is that NeuCube using EEG data can correctly identify brain patterns corresponding to specific movements. Previous work from our lab in association with research collaborators has indicated the potential of NeuCube to identify different EEG patterns relating to different imagined movements from a commercially available 14 channel EEG headset. In this trial imagined wrist extension, rest and wrist flexion achieved accuracy in 1 individual of 88%, 83% and 71% respectively (Chen et al., 2013).

The specific hypothesis for this study was that the NeuCube would accurately classify both single joint real and imagined movements of the hand into one of three classes: flexion, extension, or rest. This paradigm built on the earlier work in Chen et al. (2013) by increasing the complexity of the task in requiring the NeuCube to distinguish three conditions, two different muscle contraction patterns (flexion or extensor muscle activity) or rest. A secondary hypothesis was that the NeuCube would perform better than other classification methods, including Multiple Linear Regression (MLR), Support Vector Machine (SVM), Multilayer Perceptron (MLP) and Evolving Clustering Method (ECM) (Q. Song & Kasabov, 2001), along with offering other advantages such as adaptability to new data on-line and interpretability of results.

A.2.2 Experimental Design

Participants

Three healthy volunteers from our laboratory group participated in the study. None had any history of neurological disorders and all were right handed.

Protocol

All measures were taken in a quiet room with participants seated in a dining chair. The task consisted of either performing the specified movements or imagining the movements, or remaining at rest. All tasks were completed with eyes closed to reduce visual and blink related artifacts. The movement execution task involved the participant resting, flexing the wrist or extending the wrist. The starting position was from mid-pronation with the forearm resting on the person's lap. The movement intention task involved the participant imagining or performing the movements as described above. Participants were required to imagine or perform each movement in 2 seconds and to repeat that 10 times.

Data acquisition

A low-cost commercially available wireless Emotiv Epoc[1] EEG Neuroheadset was used to record EEG data. The Epoc records from 14 channels based on International 10-20 locations (AF3, F7, F3, FC5, T7, P7, O1, O2, P8, T8, FC6, F4, F8, AF4). Two additional electrodes (P3, P4) were used as reference. These electrodes are marked with a red dotted ring in Figure 2.1. The electrode data was digitized at 128 Hz sampling rate and sent to the computer via Bluetooth. An important factor was that other than the Emotiv's 1000 Hz high-pass filter, no filtering was applied to the data, either online or offline. This filter was only applied as it is not possible to disable it in this device.

Data processing

The data was separated into classes denoting each task. Each set of ten samples was then evenly divided into a training (seen) and a testing (unseen) set. The data was then converted into trains of spikes (one train per channel, 14 in total) with

[1] https://emotiv.com/epoc.php

the Threshold-Based Temporal Difference (TD) algorithm, $\vartheta = 6$. No other data preprocessing was applied.

Classification

Each training sample was presented to the NeuCube once, entered as 14 input streams of continuous data collected at 128 Hz, encoded using TD with a step size of 6. The spiking activity of every neuron was recorded over the time of the sample, and these presented to the deSNN classifier. The deSNN was initialized with a Mod of 0.9 and drift factor of 0.25 (empirically established values for this dataset). The synaptic weights for both the NeuCube and the deSNN were then fixed at their final (learned) values for the validation phase. The unseen data samples were presented in the same way, and the predicted classes recorded. The predicted classes were then compared to the actual classes of those samples.

Comparative Study

The NeuCube described above was compared to some of the classical machine learning methods mentioned in Chapter 2: MLR, SVM, MLP and ECM. The SVM method uses a Polynomial kernel with a rank 1; the MLP uses 30 hidden nodes with 1000 iterations for training. The ECM uses $m = 3$; $R_{\text{max}} = 1$; and $R_{\text{min}} = 0.1$. Data for these methods is averaged at 8 ms intervals and a single input vector is formed for each session, as is general practice.

NeuCube Architecture

The NeuCube architecture designed for this task followed the principles established in Chapter 5, and in particular, Section A.1.1 of this chapter. The reservoir was comprised of 1471 LIF neurons, each representing around 1 cm^3 arranged equidistant from each other within the volume of the Talairach space. Data input locations were selected based on the mapping discussed in Section A.1.1. Small world connectivity (*i.e.* distance-dependent probabilistic connectivity) was used.

A.2.3 Results

Classification accuracy for the the NeuCube averaged 76%, with individual accuracies ranging between 70-85%. There was a consistent rate of recognition between the real and the imagined movement. In terms of the comparison with other classification approaches, it is clear from the results shown in Table A.2 that the NeuCube

Subject/Session	MLR	SVM	MLP	ECM	NeuCube
1-Real	55	69	62	76	80
1-Imagined	63	68	58	58	80
2-Real	55	55	45	52	67
2-Imagined	42	63	63	79	85
3-Real	41	65	41	45	73
3-Imagined	53	53	63	53	70
Average (appr.)	52	62	55	61	76

Table A.2: Results of the comparative study for imagined limb movement captured with EEG. Accuracy is expressed as percentage correctly classified for real and imagined movements.

performed significantly better than the other machine learning techniques with the highest average accuracy over all subjects and samples, whilst the closest competitor was SVM with the second highest average accuracy of 62%. MLR was the poorest performing, with an average accuracy of 50.5%, or just over the chance threshold.

A.2.4 Discussion

This was a feasibility study to investigate the potential of using NeuCube in BCI based rehabilitation devices. In considering the classification accuracies, which ranged from 70-85%, it is important to consider three factors. Firstly, the data were collected in an unshielded room using a commercially available gaming EEG headset, resulting in an EEG signal with relatively high signal to noise ratio. Secondly, as there was no processing or feature extraction performed on the data prior to classification, the raw, noisy, EEG data was used as the input. Thirdly, all comparative methods in this study, excepting NeuCube, were validated using Leave-One-Out (all but one sample used for training), while the NeuCube was validated with a more disadvantageous 50/50 (half used for training, half for testing) split. The accuracy of the NeuCube was still significantly higher than the other techniques and would likely rise when trained with leave-one-out paradigms.

Bearing these three factors in mind the classification accuracies obtained using NeuCube are in a similar range to those reported in other research. These results demonstrate that NeuCube is capable of accurately classifying noisy and relatively low-quality data. In addition, unlike many other approaches NeuCube does not

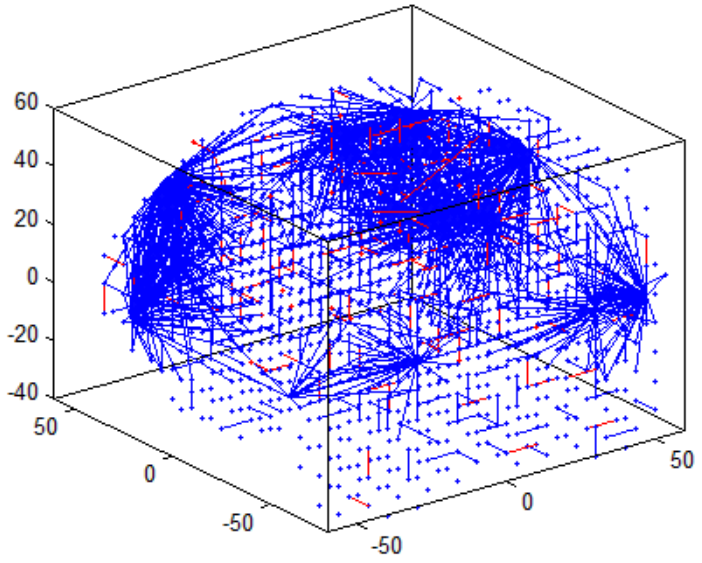


Figure A.1: Example visualisation of the connectome of the trained NeuCube. Blue lines show strong excitatory connections between two neurons, and red strong inhibitory.

require a lengthy feature extraction process, instead using all the raw data for classification, thus utilizing a rich data set that does not lose any potentially useful data.

We chose to use a relatively cheap and accessible EEG headset because two major factors that prevent the adoption of high technology interventions into rehabilitation practice are cost and complexity. EEG systems commonly used in research and clinical situations are expensive and unlikely to be widely available to rehabilitation specialists. The Emotiv neuroheadset has a limited number of channels with a fixed electrode placement, which may serve to improve usability as it reduces the preparation time and is easy for subjects to use without assistance.

An advantage of the NeuCube is that it allows for interpretation of results and understanding of the data and the brain processes that generated it. This is illustrated in Figure A.1 where the connectivity of a trained reservoir is shown for further analysis. The SNN reservoir and the deSNN classifier have evolvable structures, *i.e.* a NeuCube model can be trained further on more data and recalled on new data, which may not necessarily be of the same size or feature dimensionality. In theory, this allows for a NeuCube to be partially trained on highly accurate data captured in a controlled manner with medical-grade devices, and then further trained and adapted to the particular subject with a cheaper, less accurate consumer-grade device such as

the Emotiv. This increases the system's potential uses in clinical and rehabilitation applications.

The large number of parameters requiring optimisation limits the current NeuCube. The results presented in this study are obtained through manual parameter optimization. To mitigate this, adaptive and evolutionary techniques (including the CNGM discussed prior and quantum-inspired optimization) are being developed for this system, so that parameter selection is automated in a desirable way.

A.3 Application of Neuromorphic Systems

As briefly discussed earlier, this particular case study is an appropriate starting point for a BCI or neurorehabilitation device. In that case, the software simulation of such a system may be a limiting factor, in terms of power consumption and real-time operation. Here, we could instead consider the use of a neuromorphic hardware implementation, such as that introduced in Section 7.4. A neuromorphic implementation of this system has the potential to open up a number of applications which would otherwise be infeasible due to power or space constraints. Here, we could directly embed a low-power, real-time version of the NeuCube system into devices such as prostheses or rehabilitation exoskeletons, electric wheelchairs, *etc.*. The advantages of using a neuromorphic solution in this space are well covered in Chapter 7. The advantages of a NeuCube-based system implemented in neuromorphic hardware in such a context are primarily:

1. Significantly lowered power allows for much longer operation time in battery-powered or otherwise power constrained applications;
2. Guaranteed real time operation allows for systems to interact naturally with the user or the real world, without latency. In cases of neurorehabilitation, latency can actually inhibit neuroplasticity as the physical stimuli is then uncoupled from the mental stimuli; and, perhaps most significantly,
3. The ability for the system to continuously learn from the changing signals of the user allows for the system to continue operating successfully as the user rehabilitates or otherwise shows task-specific adaptation, where traditional methods are unable to adapt further from their initial training.

This is, of course, a non-exhaustive list of beneficial characteristics neuromorphic systems may have in this context. It is therefore suggested that future explorations of neurorehabilitation and BCI tasks in the NeuCube context should incorporate a

neuromorphic solution as early as possible, so as to ensure that these advantages are introduced as early as possible in the process.

A.4 Appendix Summary and Conclusion

In Chapter 2, we introduced some basic concepts of neuroinformatics and neuroimaging, including the most popular forms of these – EEG and fMRI. Here, we have drawn on this knowledge in order to inform our NeuCube system design. In developing a NeuCube architecture to process such data, we can draw inspiration from its implicit internal structure, to inform the 3D structure of our reservoir. This information contains spatial information which can be extracted and mapped into the space of the reservoir, by associating the data source with an existing brain structure atlas. Here, we have used the Talairach atlas, but we could as easily have used the MNI system. Once we have our reservoir structure, we can utilise our knowledge of the data collection context – *e.g.* in the case of EEG, which input locations in the International 10-20 system were used – and map these into the Talairach space. This reconciles our inputs with their respective physical locations in the model, retaining the implicit spatio-temporal links within the data. Basic templates for EEG, fMRI, and DTI data have been established here. The latter, while not a temporal source of data, can be used to inform the connectome structure of models for other types of brain data.

Additionally, in this chapter we have provided some empirical evidence for the efficacy of this approach, and the methodologies introduced in earlier chapters. Here, it is shown that a NeuCube-based SNN can effectively classify complex motor imagery data collected from a consumer-grade EEG headset. This may lead to meaningful outcomes in BCI or neurorehabilitation, among other applications.

Contributions of this Appendix

1. An introductory review of the most common forms of neuroinformatics and neuroimaging data.
2. The introduction of specific NeuCube design considerations for applications in general spatio-temporal data, and particularly in neuroinformatics data.
3. Base architectures for the classification of neuroinformatics data using the Talairach and Montreal Neurological Institute atlases for
   (a) EEG,
   (b) fMRI, and
   (c) DTI for connectome data.

4. Empirical evidence of this system's ability to effectively classify complex spatio-temporal data – in this case, motor imagery data collected with EEG.

Peer Reviewed Literature With Direct Contributions from this Appendix

1. Kasabov, N., **Scott, N. M.**, Tu, E., Marks, S., Sengupta, N., Capecci, E., Othman, M., Doborjeh, M., Murli, N., Hartono, R., Espinosa-Ramos, J.I., Zhou, L., Alvi, F., Wang, G., Taylor, D., Feigin, V., Gulyaev, S., Mahmoud, M., Hou, Z.-G. and Yang, J. (2016). Evolving Spatio-Temporal Data Machines Based on the NeuCube Neuromorphic Framework: Design Methodology and Selected Applications. *Neural Networks*. Special Issue on Learning in Big Data. Elsevier. doi:10.1016/j.neunet.2015.09.011
2. Taylor, D., Chamberlain, J., Signal, N., **Scott, N. M.**, Kasabov, N., Capecci, E., Tu, E., Saywell, N., Chen2013, Y., Hu, J., and Hou, Z.-G. (2015). Brain-Computer Interfaces for Neuro Rehabilitation. In *13th International Conference on Neuro-Computing and Evolving Intelligence*. February 19–20, Auckland, New Zealand.
3. Hu, J., Hou, Z.-G., Chen, Y., Kasabov, N., and **Scott, N. M.** (2014). EEG Based Classification of Upper-Limb ADL Using SNN for Active Robotic Rehabilitation. In *5th IEEE RAS/EMBS International Conference on Biomedical Robotics and Biomechatronics*. August 12–15, São Paulo, Brazil. IEEE. doi:10.1109/BIOROB.2014.6913811
4. Taylor, D., **Scott, N. M.**, Kasabov, N., Tu, E., Capecci, E., Saywell, N., Chen2013, Y., Hu, J., and Hou, Z.-G. (2014). Feasibility of the NeuCube SNN architecture for detecting motor execution and motor intention for use in BCI applications. In *Proceedings of the IEEE International Joint Conference on Neural Networks*. Beijing, China. IEEE. doi:10.1109/IJCNN.2014.6889936
5. Kasabov, N., Hu, J., Chen2013, Y., **Scott, N. M.**, and Turkova, Y. (2013). Spatio-temporal EEG data classification in the NeuCube 3D SNN Environment: Methodology and Examples. In *Proceedings of the 20th International Conference on Neural Information Processing*, 3–7 November 2013, Daegu, Korea. Springer. doi:10.1007/978-3-642-42051-1_9
6. **Scott, N. M.**, Kasabov, N., and Indiveri, G. (2013). NeuCube Neuromorphic Framework for Spatio-Temporal Brain Data and Its Python Implementation. In *Proceedings of the 20th International Conference on Neural Information Processing*, 3–7 November 2013, Daegu, Korea. Springer. doi:10.1007/978-3-642-42051-1_11

Appendix B

Case Study in Radioastronomy

The previous chapter introduced a case study showing the NeuCube's effectiveness on spatio-temporal data. Here, its application to spectro-temporal data will be introduced. In particular, we introduce it in the (very preliminary) context of high-speed streaming radioastronomy data.

We are interested in spectro-temporal data as it, like spatio-temporal data, contains an intrinsic relationship between its temporal dynamics and its structure. In the sense of structure, we mean its spectral makeup, not its spatial distribution. Interestingly, in the NeuCube, we can map the spectral structure to a physical structure, in order to represent it effectively in the reservoir.

Here, we will take the example of a radioastronomical signal – *i.e.*, one that has been recorded by a radiotelescope. Recall from Section 2.2 the general context of a radioastromonical signal. In particular, we are primarily interested in addressing the issue of identifying pulsars in radiotelescope data.

B.1 NeuCube Architecture for Spectro-Temporal Data

Developing a reservoir architecture for spectro-temporal data is somewhat more complex than developing one to represent spatio-temporal data. Primarily this is due to the obvious fact that there is generally no inherent spatial structure to the data, and therefore, no obvious way to represent that in the positions of the reservoir neurons.

However, it is possible to represent some spectral characteristics of the data as spatial characteristics. In a general sense, we use our *a-priori* knowledge of the data source

to create a conceptual model of the spectral properties. This is a heuristic process, and is heavily dependent on the actual data source.

Here, we take the example of a radioastronomical signal. As the data does not represent a spatial structure, it is not directly possible to create a meaningful reservoir shape informed by it. In a case such as this, we generate an arbitrary shaped reservoir – in this case, a cuboid.

Recall from our earlier discussion that a radioastronomical signal such as that received from a pulsar has a characteristic relationship between the relative time signal's component frequencies arrival at the sensor, and the distance that signal travelled. In general, this can be modelled as an inverse-quadratic. That is, the lower frequencies arrive at the sensor later, due to their dispersal through interstellar media.

This inverse quadratic shape is an intrinsic characteristic of the relationship between the spectral components and the temporal components of this data. Therefore, it makes sense for us to represent this shape in the NeuCube reservoir as best we can. The most efficient representation in terms of space in a 3D reservoir is to associate each spectral channel with a single input neuron, aligned diagonally across x and y axes the reservoir and (viewed as a horizontal slice in the z-axis) proportionally representing the inverse quadratic shape.

In this way, we can – to some extent – represent the *spectral* nature of the data in a *spatial* manner. Obviously, this type of mapping will always be an approximation, but it does still adhere to the key principle of a NeuCube reservoir's structure; a representation of the data that retains the implicit spatio-temporal links within a data source.

This principle extends to sources other than radiostronomy data. The identification of the most meaningful structure in the data based on our *a-priori* knowledge of what the data represents is key in designing a meaningful reservoir structure. In the unlikely case that no inherent structure can be identified in the data, we can instead use some arbitrary shaped reservoir in place, with the proviso that the design process follows that identified in Chapter 5.

B.2 Dispersed Transient and Pulsar Search

Here, we introduce a proof-of-concept case study which sought to identify whether the NeuCube was appropriate for application on general spectro-temporal data, in the context of radioastronomy data.

The study in this section was first introduced in Scott et al. (2015).

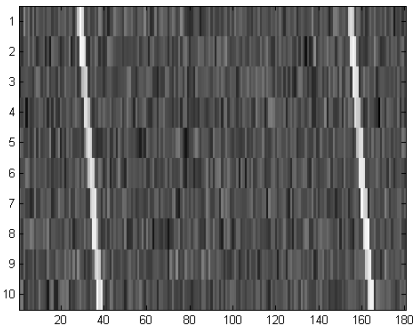
B.2.1 Motivation & Research Questions

With the introduction of the Square Kilometre Array Project, a revolution in the data available to radioastronomers is occurring. Of particular interest is the identification of distinctive spectral patterns known as dispersed transients (single, bright pulses of unknown extraterrestrial origin) or dispersed pulsars (characteristic signals given off by the rotation of pulsar stars). These signals, if identified and analysed correctly, can have major implications towards our understanding of relativistic physics, and therefore, our understanding of the fundamental forces at work in our universe. However, these signals are highly infrequent (class imbalance of 1:10,000-12,000 pulsar to noise), highly unpredictable in terms of signal characteristics, and buried in noise. The current state of the art approach requires a brute force search in terms and is untenable in the face of the volume of data the SKA will produce – a data stream rate of 1.5-2.5 TBps.

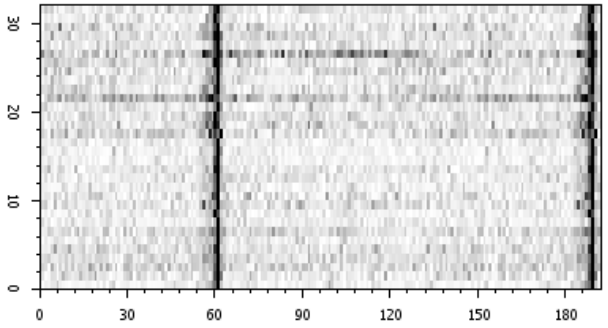
An alternative approach using neuromorphic principles (the NeuCube evolving spatio-temporal data machine) as a first line candidate selection system is therefore proposed here. This is appropriate as NeuCube eSTDM provides compact representation of spatial, spectral, and temporal characteristics, evolving learning, non-linear pattern recognition, and low computation cost comparative to alternative techniques, particularly when implemented on neuromorphic hardware.

B.2.2 Experimental Design

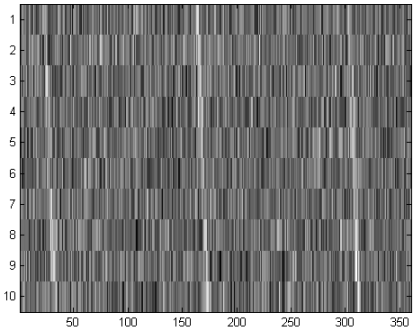
A synthetic dataset was generated, containing 30 positive (pulsar + noise) and 30 negative (solely noise) samples. The code used to generate this simulated data is given in Appendix G. The generated noise is a zero-mean Gaussian with $\sigma = 1$, in accordance with the type of noise found in real observations. The strength, periodicity, and dispersion of this pulsar signal is varied randomly between samples, representing 25 distinct pulsar candidates. The dispersion follows the standard inverse quadratic shape. Signal-to-noise ratio ('strength') of these candidates varied randomly in the range beween 3:1–40:1. Periodicity was randomly selected in the range between 200–500 milliseconds. The offset of the first pulse from the start of the observation was selected randomly in the range between 15–40 milliseconds. Additionally, the width of the pulse itself was selected in the range between 2–5 milliseconds.



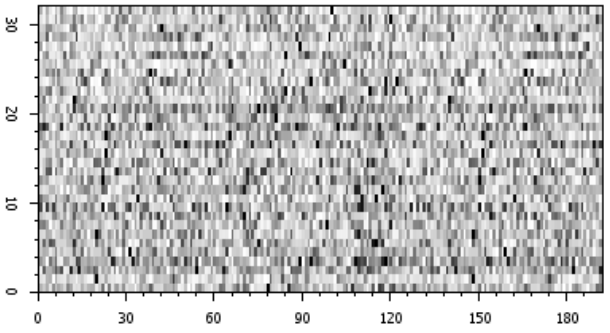
(A) Strong synthetic pulsar



(B) Strong real pulsar



(C) Weak simulated pulsar



(D) Weak real pulsar

FIGURE B.1: Example of the synthetic radioastronomy data compared with the real data. The y-axis represents the frequency band, and the x-axis is time. On the left we can see the synthetic (generated) data, which is similar to the real plots shown on the right hand side. The difference in dispersion of the plots is due to fact that the real data was plotted after dedispersion. The raw data is, however, similar.

A visualisation of this synthetic data compared to the real data is given in Figure B.1. The top two figures show a 'strong' synthetic (Figure B.1a) and real (Figure B.1b) pulsar, with a SNR of ≈ 40. The bottom two figures show a 'weak' synthetic (Figure B.1c) and real (Figure B.1d) pulsar, with a SNR of ≈ 3. Both are viable pulsar candidates that should be identified by our system. As is clear from these plots, it can be difficult for a human observer to identify a pulsar in such data, particularly at the rates we have previously discussed.

This dataset was separated randomly using the holdout method (2-fold cross validation), *i.e.* that it is divided into two equal datasets (here named A and B) for training and testing. The NeuCube was trained on one of these divisions (A) and validated on the other (B). The second fold inverted the order (train on B and validate on A). Each sample was presented once to the network.

A grid search was utilised to find the optimal network design. The parameters optimised for were the threshold for the encoding scheme (TD), the reservoir neurons'

spiking thresholds, the small-world connection probability, the A (magnitude) of the STDP implementation, and the deSNN parameters.

NeuCube Architecture

The NeuCube architecture used was a $10 \times 10 \times 10$ cube of LIF neurons, connected with the small-world method described in Section 6.4.3.1. The input locations were designed using the principles established in Chapter 5 and Section B.1. The data was encoded using the Threshold-Based Temporal Difference (TD) algorithm, with its parameters established by the grid search process. A deSNN classifier was used as the output device.

B.2.3 Results

Positive classification accuracies were reported in a range of 84-92%, depending on the random split of training/testing data taken and the parameter selected in the grid search. Sensitivity/specificity tables indicate a bias towards classifying positive low SNR samples as noise.

B.2.4 Discussion

While any meaningful conclusions cannot be drawn from such a preliminary experiment, there is certainly enough evidence here for the justification of further experiments. Future experiments must improve the classification accuracy, and in particular, emphasise a bias towards False Positives (noise classified as pulsar/transient candidate) rather than False Negatives, in order to maximise the potential information gained by this system. Considerations must be paid to processing speed, the speed of the palimpsest dynamic in the model, and computational cost – particularly power consumption, as these systems will be installed in a remote location.

The real dataset will be considerably more unbalanced than this synthetic set ($\approx$ 1:10,000 P:N ratio), which introduces some issues. However, it is expected that the NeuCube will be able to handle this imbalance, as it should not learn the 'noise' patterns in the same way as a classical ML algorithm might – these should be integrated away through the spike encoding and internal learning processes.

At present, the simulations have been performed in MATLAB. This is inappropriate for a production system as it is incapable of dealing with the data throughput, and requires a commodity computing platform. In the future, we will apply the SpiNNaker

version of the NeuCube in this domain. This dedicated hardware will ensure the scaling capacity and constant time for computation regardless of NeuCube network size, at much lower power consumption than a software emulation.

These very preliminary experiments suggest that further work is warranted, as the results (particularly with respect to the low SNR samples) suggest that this approach may prove to be an effective approach to the challenge of automated pulsar candidate search.

Experiments are planned to iteratively add complexity. We intend to beginn with strong transients, over a limited number of channels, then progress to strong transients with the full SKA channels, increasing class disparity and reducing SNR to real levels, weak transients with limited channels, and so on. The same order will be repeated for pulsar search. Due to the scale of the data and the models used, these will be applied on the SpiNNaker neuromorphic computation system.

Additionally, we will be collaborating with W. van Sträten of Swinburne University of Technology. His group will be providing us with access to $\approx$ 900 TB of radioastronomical survey data, which we will be utilising for a real-world study of this system's efficacy. This is, of course, subsequent to us proving its efficacy on data more complex than this proof-of-concept.

B.3 Application of Neuromorphic Systems

Application of such a NeuCube system in a neuromorphic context is ideal for the case of radioastronomy data. A particular constraint of the SKA programme discussed in Section 2.2 is that the systems will be deployed in remote geographical locations; as a result, their local power supply is limited. Power conservation is therefore a consideration for systems incorporated in the SKA. To this end, a neuromorphic implementation of the NeuCube for first-line event detection is advantageous over a commodity computing solution, as the power consumption can be limited and managed more precisely. In this case, we could use the NeuCube to identify events of interest which are worth further analysis on the more costly conventional systems which will be the primary computing devices in the SKA.

The nature of the NeuCube reservoir is such that it theoretically integrates away random noise, and emphasises temporally correlated signals; in this way, it is likely an efficient option for the implementation of a learning system in the context of

radioastronomy signals, which are necessarily temporally correlated signals buried in random noise.

Additionally, the real-time performance guarantees of a neuromorphic solution are advantageous in this context. Here, we can be sure that the information streamed in will be processed in real time; a necessary feature in developing systems for the SKA, as no sensor data will be recorded long-term. Instead, key areas of interest are broadly highlighted in real time during an initial survey, to be revisited in more depth in later studies. Here, a real-time solution is implicitly necessary.

A NeuCube implementation in neuromorphic hardware is therefore a natural progression of this study. It is theoretically advantageous in terms of power consumption, processing time, and learning ability over conventional methods, and should therefore be explored further in later works.

B.4 Appendix Summary and Conclusion

Application of the NeuCube framework to a spectro-temporal task – in this case, the classification of complex radioastronomy data – has been introduced. Firstly, we reviewed the basic precepts of radioastronomy data collection, particularly in the context of the current SKA project. One of the major aims of this project is the automated identification of a specific class of pulsar, which can be used to infer some characteristics of relativistic physics. The challenge here, is that this data is highly complex, noisy, slowly shifting, and is extremely rare in a massive data stream. The NeuCube is a theoretical match for this data source. Herein, we established a proof-of-concept case study for a simplified form of this data source, which shows good promise.

Also introduced were some concepts for the design of NeuCube architectures in a spectro-temporal context. We can represent the inherent structure of the spectral component of the data spatially, by taking some characteristic of the data signal and mapping this to an SNN reservoir. In the case study, we take the most significant spectral structure – the dispersion of the frequency bands due to interstellar media – and map this to the spatial structure of the reservoir. This method is generalisable to other forms of spectro-temporal data.

Contributions of this Appendix

1. The introduction of a general methdology for the development of NeuCube architectures for spectro-temporal data
2. The introduction of a novel NeuCube architecture for radioastronomical data
3. A novel approach to radioastronomy radio-frequency interference identification, with advantages in:
   (a) Speed
   (b) Computational cost
   (c) Power consumption
   (d) Adaptiveness
4. Empirical evidence of the effectiveness of this novel NeuCube architecture applied to general spectro-temporal data, in the specific case of radioastronomical data
5. A future direction of research for pulsar and RFI identification using neuromorphic principles

Peer Reviewed Literature With Direct Contributions from this Appendix

1. Kasabov, N., **Scott, N. M.**, Tu, E., Marks, S., Sengupta, N., Capecci, E., Othman, M., Doborjeh, M., Murli, N., Hartono, R., Espinosa-Ramos, J.I., Zhou, L., Alvi, F., Wang, G., Taylor, D., Feigin, V., Gulyaev, S., Mahmoud, M., Hou, Z.-G. and Yang, J. (2016). Evolving Spatio-Temporal Data Machines Based on the NeuCube Neuromorphic Framework: Design Methodology and Selected Applications. *Neural Networks*. Special Issue on Learning in Big Data. Elsevier. doi:10.1016/j.neunet.2015.09.011
2. **Scott, N. M.**, Mahmoud, M., Hartono, R., Gulyaev, S., and Kasabov, N. (2015). Feasibility analysis of using the NeuCube Spiking Neural Network Architecture for Dispersed Transients and Pulsar Detection. In *13th International Conference on Neuro-Computing and Evolving Intelligence*. February 19–20, Auckland, New Zealand.

Appendix C

Listing of Significant Classes for a NeuCube Implementation in PyNN

The code listings here are current at the time of publication. A version of the code has been frozen at this time and has been made available at https://github.com/KEDRI-AUT/NeuCube_PyNN/releases/tag/thesis_freeze. Up-to-date versions are available in a Git repository found at https://github.com/KEDRI-AUT/NeuCube_PyNN. For access, please contact the author via email (nathan.scott@aut.ac.nz) or see Appendix D for details. No warranty as to the operation of this code on your system is made; please, contact the author if you have issues or check the Github link for an updated version.

C.1 Main

```
1  import JSONLoader
2  import argparse
3  from NeuCubeReservoir import *
4  from spike_encoders.TemporalContrastEncoder import *
5  from classifiers.DynamicEvolvingSNNClassifier import *
6
7
8  parser = argparse.ArgumentParser()
9  parser.add_argument("json_file", help="The input file")
10 args = parser.parse_args()
11 configuration_loader = JSONLoader
12 configuration_data = None
13 if args.json_file is not None:
14   configuration_data = configuration_loader.load_JSON(args.json_file)
15 else:
16   raise LookupError("Configuration file not found. Please call main.py
       *filename* with a JSON formatted file.")
17
18 file_prefix = 'model_data/'
19
20 encoder = TemporalContrast()
```

```
input_spikes = encoder.encode_spikes(configuration_data["data"]["
    input_directory"])

ncr = NeuCubeReservoir(configuration_data["reservoir"],
    configuration_data["location_prefix"])
spike_times, class_labels = ncr.run_sim(input_spikes,
    configuration_data["data"], configuration_data["run"])

classifier_data = dict(zip(class_labels, spike_times))
classifier = DynamicEvolvingSNNClassifier()
classifier.classify(classifier_data, split=0.8)
```

Listing C.1: Implementation of the main control loop of the application

C.2 NeuCubeReservoir

```
import pyNN.spiNNaker as p
import random as rand
from NetworkStructure import NetworkStructure
from time import strftime
from Plot import Plot


class NeuCubeReservoir:
  def __init__(self, configuration_dict=None, location_prefix="
    model_data/"):
    self.configuration_dict = configuration_dict
    self.location_prefix = location_prefix
    if self.configuration_dict is not None:
      self.structure_file_name = self.location_prefix + self.
    configuration_dict["model_data_location"]
      self.input_location_file_name = self.location_prefix + self.
    configuration_dict["input_data_location"]
      self.connection_probability_factor = self.configuration_dict["
    p_connection"]
    self.spike_times = []
    self.class_labels = {}

  def run_sim(self, data, data_dict=None, run_dict=None):
    """
    Sets up and runs the simulation
    :param data_dict:
    :param run_dict:
    """
    num_runs = int(run_dict["times_to_run"])
    self.class_labels = data_dict["class_labels"]

    num_samples = data_dict["n_samples"] # number of samples to learn
    sim_time = run_dict["sim_time"]  # time to run sim for
    inhibitory_split = 0.2
    plot_spikes = run_dict["plot_spikes"]
    save_figures = False
    show_figures = run_dict["show_plots"]
    sim_start_time = strftime("%Y-%m-%d_%H:%M")

```

```
36      cell_params_lif = self.configuration_dict["cell_parameters"]
37
38      # Create the 3d structure of the NeuCube based on the user's given
        structure file
39      network_structure = NetworkStructure()
40      if self.structure_file_name is not None:
41        network_structure.load_structure_file(filename=self.
        structure_file_name)
42      else:
43        network_structure.load_structure_file()
44      num_neurons = len(network_structure.get_positions())
45      if self.input_location_file_name is not None:
46        network_structure.load_input_location_file(filename=self.
        input_location_file_name)
47      else:
48        network_structure.load_input_location_file()
49      # Calculate the inter-neuron distance to be used in the small world
         connections
50      network_structure.calculate_distances()
51      # Generate two connection matrices for excitatory and inhibitory
        neurons based on your defined split
52      network_structure.calculate_connection_matrix(inhibitory_split,
        self.connection_probability_factor)
53      # Get these lists to be used when connecting the neurons later
54      excitatory_connection_list = network_structure.
        get_excitatory_connection_list()
55      inhibitory_connection_list = network_structure.
        get_inhibitory_connection_list()
56      # Choose the correct neurons to connect them to, based on your a-
        priori knowledge of the data source -- eg, EEG
57      # to 10-20 locations, fMRI to voxel locations, etc.
58      input_neuron_indexes = network_structure.find_input_neurons()
59      num_inputs = len(input_neuron_indexes)
60      # Make the input connections based on this new list
61      input_weight = 4.0
62      input_connection_list = []
63      for index, input_neuron_index in enumerate(input_neuron_indexes):
64        input_connection_list.append((index, input_neuron_index,
        input_weight, 0))
65
66      for run_number in xrange(num_runs):
67        excitatory_weights = []
68        inhibitory_weights = []
69        for sample_number in xrange(num_samples):
70          # At the moment with the limitations of the SpiNNaker hardware
        we have to reinstantiate EVERYTHING
71          # each run. In the future there will be some form of repetition
         added, where the structure stays in memory
72          # on the SpiNNaker and only the input spikes need to be updated
        .
73
74          # data_prefix = sim_start_time + "_r" + str(run_number + 1) +
        "-s" + str(sample_number + 1)
75
76          # Set up the hardware - min_delay should never be less than the
         timestep.
```

```
77          # Timestep should = 1.0 (ms) for normal realtime applications
78          p.setup(timestep=1.0, min_delay=1.0)
79          p.set_number_of_neurons_per_core("IF_curr_exp", 100)
80
81          # Create a population of neurons for the reservoir
82          neurons = p.Population(num_neurons, p.IF_curr_exp,
    cell_params_lif, label="Reservoir")
83
84          # Setup excitatory STDP
85          timing_rule_ex = p.SpikePairRule(tau_plus=20.0, tau_minus=20.0)
86          weight_rule_ex = p.AdditiveWeightDependence(w_min=0.1, w_max
    =1.0, A_plus=0.02, A_minus=0.02)
87          stdp_model_ex = p.STDPMechanism(timing_dependence=
    timing_rule_ex, weight_dependence=weight_rule_ex)
88          # Setup inhibitory STDP
89          timing_rule_inh = p.SpikePairRule(tau_plus=20.0, tau_minus
    =20.0)
90          weight_rule_inh = p.AdditiveWeightDependence(w_min=0.0, w_max
    =0.6, A_plus=0.02, A_minus=0.02)
91          stdp_model_inh = p.STDPMechanism(timing_dependence=
    timing_rule_inh, weight_dependence=weight_rule_inh)
92
93          # Generate a population of SpikeSourceArrays containing the
    encoded input spike data
94          spike_sources = p.Population(num_inputs, p.SpikeSourceArray, {'
    spike_times': data[sample_number]})
95          # or you can use poisson trains for examples
96          # spike_sources = p.Population(num_inputs, p.SpikeSourcePoisson
    , {'rate': rand.randint(20, 60)},
97          #               label="Poisson_pop_E")
98
99          # Connect the input spike sources with the "input" neurons
100         connected_inputs = p.Projection(spike_sources, neurons, p.
    FromListConnector(input_connection_list))
101
102         # If we have weights saved/recorded from a previous run of this
     network, load them into the structure
103         # population.set(weights=weights_list) and population.
    setWeights(weight_list) are not supported in
104         # SpiNNaker at the moment so we have to do this manually.
105         if excitatory_weights and inhibitory_weights:
106           for index, ex_connection in enumerate(
    excitatory_connection_list):
107             ex_connection[2] = excitatory_weights[index]
108           for index, in_connection in enumerate(
    inhibitory_connection_list):
109             in_connection[2] = inhibitory_weights[index]
110
111         # Setup the connectors
112         excitatory_connector = p.FromListConnector(
    excitatory_connection_list)
113         inhibitory_connector = p.FromListConnector(
    inhibitory_connection_list)
114
115         # Connect the excitatory and inhibitory neuron populations
```

```
116         connected_excitatory_neurons = p.Projection(neurons, neurons,
    excitatory_connector,
117                                 synapse_dynamics=p.SynapseDynamics(slow=
    stdp_model_ex),
118                                 target="excitatory")
119         connected_inhibitory_neurons = p.Projection(neurons, neurons,
    inhibitory_connector,
120                                 synapse_dynamics=p.SynapseDynamics(slow=
    stdp_model_inh),
121                                 target="inhibitory")
122 
123         # Set up recording the spike trains of all the neurons
124         neurons.record()
125         spike_sources.record()
126 
127         # Run the actual simulation
128         p.run(sim_time)
129 
130         # Save the output spikes
131         spikes_out = neurons.getSpikes(compatible_output=True)
132         self.spike_times.append(spikes_out)
133         input_spikes_out = spike_sources.getSpikes(compatible_output=
    True)
134         # Get the synaptic weights of all the neurons
135         excitatory_weights = connected_excitatory_neurons.getWeights()
136         inhibitory_weights = connected_inhibitory_neurons.getWeights()
137 
138         # when we're all done, clean up
139         p.end()
140 
141         # Make some plots, save them if required. Check if you need to
    either save or show them, because if not,
142         # there's no point wasting time plotting things nobody will
    ever see.
143         if plot_spikes and (save_figures or show_figures):
144           plot = Plot(save_figures, data_prefix)
145           # Plot the 3D structure of the network
146           plot.plot_structure(network_structure.get_positions(),
    figure_number=0)
147           # Plot the spikes
148           plot.plot_spike_raster(spikes_out, sim_time, num_neurons,
    figure_number=1)
149           # Plot the weights
150           plot.plot_both_weights(excitatory_weights, inhibitory_weights
    , figure_number=2)
151           # If we want to show the figures, show them now, otherwise
    ignore and move on
152           if show_figures:
153             # Show them all at once
154             plot.show_plots()
155           plot.clear_figures()
156           plot = None
157     return self.spike_times, self.class_labels
158 
159 if __name__ == "__main__":
160   ncr = NeuCubeReservoir()
```

```
161   ncr.run_sim()
```

Listing C.2: Implementation of a generic NeuCube 3D reservoir

C.3 NetworkStructure

```
1  import csv
2  import random as rand
3  import scipy.spatial as scipy_spatial
4  import numpy as np
5  import numba
6
7  def _structure_file_load(filename):
8    """
9    Does the actual file loading/reading logic for an position file.
10   :param filename: the file to be loaded and read
11   :return: a list of 3D tuples of x,y,z locations
12   """
13   csv_file = open(filename, 'r')
14   reader = csv.reader(csv_file, delimiter=',')
15   # Read neuron positions from file
16   positions = []
17   for line in reader:
18     p = (float(line[0]), float(line[1]), float(line[2]))
19     positions.append(p)
20   return positions
21
22
23 class NetworkStructure(object):
24   """
25   This class contains methods to load a 3D neuron structure, calculate
       inter-neuron distances and generate connectomes
26   based on these distances. It will eventually be extended to
       incorporate other connectome generation methods.
27   """
28
29   def load_structure_file(self, filename='model_data/neuron_positions.
       txt'):
30     """
31     Load and read the file containing the 3D structure of a NeuCube
       reservoir
32     :param filename: the filename of the structure
33     """
34     self.positions_list = _structure_file_load(filename)
35
36   def load_input_location_file(self, filename='model_data/
       input_positions.txt'):
37     """
38     Load and read the file containing the desired 3D locations of the
       input neurons.
39     :param filename: the filename of the structure
40     """
41     self.input_positions = _structure_file_load(filename)
42
43   def calculate_distances(self):
```

```
    """
    Calculate the distances from the loaded 3D locations file.
    This method should check if a saved version of the distances for the given dataset already exists, and only
    calculate these if it does not. It should then save a file containing them. This saves time on large datasets.
    """
    if not self.positions_list:   # if the file hasn't already been loaded
      self.load_structure_file() # load it!
    pos_list = np.array(self.positions_list)
    self.distances = scipy_spatial.distance.pdist(pos_list, metric='euclidean')
    self.max_distance = np.amax(self.distances)

  @numba.jit
  def calculate_connection_matrix(self, inhibitory_split=0.2, connection_probability=0.025):
    """
    Calculate the connection matrices based on distance dependent probability and a given inhibitory split.
    :param inhibitory_split: the proportion of inhibitory:excitatory neurons in the population
    """
    for i, presynaptic_pos in enumerate(self.positions_list):
      inhibitory_neuron = False
      if rand.random() < inhibitory_split:
        inhibitory_neuron = True
      for j, postsynaptic_pos in enumerate(self.positions_list):
        if i is not j:
          curr_idx = i + 2 * j
          normalised_distance = self.distances[curr_idx] / self.max_distance
          # Distance dependence from Neuronal Dynamics by Gerstner, Kistler, et. al.
          conn_prob = connection_probability * np.e ** (-normalised_distance)
          if rand.random() < conn_prob:
            # This ensures the the synaptic delay scales linearly with distance and also will never go above
            # the 16ms max delay normally implemented in SpiNNaker
            delay = normalised_distance * self.delay_factor
            if inhibitory_neuron:
              self.inhibitory_connection_list.append([i, j, rand.gauss(0.4, 0.2), delay])
            else:
              self.excitatory_connection_list.append([i, j, rand.gauss(0.5, 0.3), delay])

  def get_excitatory_connection_list(self):
    """
    Returns a list of connections for excitatory neurons.
    :return: n-length list of [presynaptic index, postsynaptic index, weight, distance dependent delay]
    """
    return self.excitatory_connection_list
```

```
86
87    def get_inhibitory_connection_list(self):
88      """
89      Returns a list of connections for inhibitory neurons.
90      :return: n-length list of [presynaptic index, postsynaptic index,
        weight, distance dependent delay]
91      """
92      return self.inhibitory_connection_list
93
94    def get_positions(self):
95      """
96      Returns the list of 3D positions of the neurons in the network
97      :return: n-length list of (x,y,z) locations
98      """
99      return self.positions_list
100
101   def get_delays(self):
102     raise NotImplementedError("Delay lookup is not implemented")
103     # return self.delays
104
105   def get_weights(self):
106     raise NotImplementedError("Weight lookup is not implemented")
107     # return self.weights
108
109   def find_input_neurons(self, query_list=None, input_neighbourhood=1):
110     """
111     Find the nearest neurons from a list of 3D points and a search
        neigbourhood, to define which neurons in the 3D
112     NeuCube reservoir should be connected to the spike sources based on
         our a-priori knowledge
113     of the data.
114     :param query_list: an n length list of [x,y,z] positions
115     :param input_neighbourhood: the number of closest neurons to return
         for each point, default = 1
116     :return: a list of lists (possibly of lists, if input_neighbourhood
         > 1) of neuron indexes in query list
117     """
118     locations = []
119     if not query_list:
120       query_list = self.input_positions
121     for location in query_list:
122       locations.append(self.find_closest_neuron(location,
        input_neighbourhood))
123     return locations
124
125   def find_closest_neuron(self, query_position, search_neigbourhood=1):
126     """
127     Find the nearest neuron from a given 3D point in a given
        neighbourhood. This is used to connect the input
128     neurons to their closest location in the 3D structure.
129     :param query_position: list [x,y,z] positions to query from
130     :param search_neigbourhood: the number of neighbours query for,
        default = 1
131     :return: a single list [x,y,z] of the nearest neuron positon, or a
        list of lists of the same
132     """
```

```
133     positions_tree = scipy_spatial.cKDTree(self.positions_list)
134     closest_neuron_index = positions_tree.query(query_position, k=
        search_neigbourhood)
135     return closest_neuron_index[1]
136
137   def __init__(self):
138     self.positions_list = []
139     self.input_positions = []
140     self.inhibitory_connection_list = []
141     self.excitatory_connection_list = []
142     self.delays = []
143     self.weights = []
144     self.distances = None
145     self.delay_factor = 16.0 # 16 ms is the max delay supported
        normally in SpiNNaker
146     self.max_distance = 0
147
148 if __name__ == "__main__":
149   net_structure = NetworkStructure()
150   net_structure.load_structure_file()
151   net_structure.calculate_distances()
152   net_structure.calculate_connection_matrix()
153   print len(net_structure.excitatory_connection_list)
154   print len(net_structure.inhibitory_connection_list)
```

Listing C.3: Manual implementation of structure and distance for the reservoir, implemented in standard Python

C.4 GenericClassifier

```
1 class GenericClassifier():
2
3   def __init__(self):
4     pass
5
6   def modify_classifier(self, **kwargs):
7     raise NotImplementedError("Do not use GenericClassifier. No
      specific classifier defined.")
8
9   def classify(self, **kwargs):
10    raise NotImplementedError("Do not use GenericClassifier. No
      specific classifier defined.")
```

Listing C.4: Superclass of any Classifiers implemented

C.5 DynamicEvolvingSNNClassifier

```
1 import GenericClassifier
2 import pyNN.spiNNaker as p
3 import random as rand
4 import scipy.spatial.distance as distance_calc
5
6
7 # Implementation based on the algorithm defined in:
```

```
8  # Dhoble, K., Nuntalid, N., Indiveri, G., and Kasabov, N. (2012).
       Online Spatio-Temporal Pattern Recognition with
9  # Evolving Spiking Neural Networks utilising Address Event
       Representation, Rank Order, and Temporal Spike Learning
10 # In Proc. WCCI 2012 IJCNN. Brisbane, Australia. IEEE.
11
12
13 class DynamicEvolvingSNNClassifier(GenericClassifier.GenericClassifier)
       :
14   """
15
16   """
17   def __init__(self):
18     GenericClassifier.__init__(self)
19     self.n_inputs = 1471 # number of input neurons to calculate weights
        for, from NeuCubeReservoir
20     self.n_classes = 2 # number of classes
21     self.n_samples = 10 # number of samples per class
22     self.t_sample = 1000 # time in ms for each sample
23     self.t_simulation = self.t_sample
24     self.deSNN_C = 0.8 # "C" variable for threshold calculations
25     self.deSNN_mod = 0.8 # "Mod" var for for PSP calculations
26     self.deSNN_sim = 10. # Similarity measure for the neuron merging
27     self.weight_list = []
28     self.psp_thresholds = []
29     self.class_labels = []
30     # there is never a leak rate in these models
31     # spiking threshold voltage has not yet been set as this needs to
       be calculated after the simulation
32     self.cell_params_out = {'cm': 0.25, 'i_offset': 0.0, 'tau_refrac':
       2.0, 'tau_syn_E': 3.0,
33                 'tau_syn_I': 3.0, 'v_reset': -65.0, 'v_rest': -65.0}
34     self.class_labels = {}
35     # all of the above variables should be taken from the config file
       or the respective feeder classes
36
37   def modify_classifier(self):
38     """
39
40     :return:
41     """
42     raise NotImplementedError("The ability to change classifier
       parameters online has not yet been implemented")
43     # capacity to change params on demand
44
45   def classify(self, data, split=0.5):
46     """
47
48     :param data:
49     :param split:
50     :return:
51     """
52     rand.shuffle(data)
53     split_num = int(len(data) / split)
54     train_data = data[:split_num]
55     self.train(train_data)
```

```
    test_data = data[split_num:]
    self.verify(test_data)

  def train(self, data, class_labels):
    """
    Basic algorithm for deSNN.
    See the paper by Dhoble, et.al. (2012) in file header for details.
    @param data:
    @return:
    """
    current_outputs = 0
    self.psp_thresholds = []
    previous_merges = []
    self.weight_list = [] # make sure it's empty
    for sample in data:
      current_outputs += 1
      previous_merges.append(0)
      p.setup() # needed for PyNN

      # the neurons providing the input signals are always the same, only their activity changes
      input_neurons = p.Population(self.n_inputs, p.SpikeSourceArray, {'spike_times': sample})

      # setup the output neuron population
      output_neurons = p.Population(current_outputs, p.IF_curr_exp, cellparams=self.cell_params_out)
      for i, neuron in enumerate(output_neurons):
        if self.psp_thresholds[i] is not None:
          neuron.set(v_thresh=self.psp_thresholds[i])

      # weight list is based on the order of the incoming spikes
      # need to go through the input spike trains and rank them
      new_weights = []
      self.weight_list.append(new_weights)

      # setup the connections
      self.input_connection_list = [[[from_neuron, output_neuron, self.weight_list[from_neuron], 1.0]
                      for from_neuron in xrange(0, self.n_inputs)]
                     for output_neuron in xrange(0, current_outputs)]

      # setup drift - here using STDP instead of SDSP because sPyNNaker does not support SDSP
      timing_rule_drift = p.SpikePairRule(tau_plus=20.0, tau_minus=20.0)
      weight_rule_drift = p.AdditiveWeightDependence(w_min=0.1, w_max=1.0, A_plus=0.02, A_minus=0.02)
      drift_model = p.STDPMechanism(timing_dependence=timing_rule_drift, weight_dependence=weight_rule_drift)
      network = p.Projection(input_neurons, output_neurons, p.FromListConnector(self.input_connection_list),
                 synapse_dynamics=drift_model)

      # add recorder for the spike times and membrane potentials
      output_neurons.record(['v', 'spikes'])
```

```
103
104       # run simulation
105       p.run(self.t_sample)
106
107       # save output weights and traces
108       # traces & spike times first:
109       data_out = output_neurons.get_data().segments[0]
110       output_neurons.get_data(clear=True)
111
112       # find max psp for each neuron
113       newest_neuron_data = data_out.filter(name="v")[0]
114       newest_neuron_data = newest_neuron_data[-self.n_inputs:]
115       max_psp = find_max(newest_neuron_data)
116
117       # calculate psp_th
118       psp_th = max_psp * self.deSNN_C
119
120       # sort out the weights - similarity comparison and potential
    merging
121       trained_weights = network.getWeights()
122       current_sample_weights = trained_weights[-self.n_inputs:] # gets
    the most recent weights
123       offset = 0
124       merged = False
125       # do merge comparison
126       for neuron_index in xrange(0, current_outputs):
127         comparing_weights = trained_weights[offset:self.n_inputs]
128         distance = distance_calc.euclidean(comparing_weights,
    current_sample_weights)
129         if distance <= self.deSNN_sim:
130           # merge the neurons, generate a new weight_list and updated
    psp_thresholds
131           self.weight_list, self.psp_thresholds[neuron_index] =
    merge_neurons(comparing_weights,
132                                         current_sample_weights,
133                                         previous_merges[neuron_index
    ],
134                                         self.psp_thresholds[
    neuron_index],
135                                         psp_th)
136           merged = True
137           break
138         offset += self.n_inputs
139
140       if not merged:
141         # now we know we need all of them, we save the parameters:
142         self.weight_list = trained_weights
143         self.psp_thresholds.append(psp_th)
144
145       self.class_labels["{0}".format(current_outputs)] = class_labels[
    current_outputs]
146
147       p.end() # needed for PyNN
148
149   def verify(self, data):
150     """
```

```
151
152     :param data:
153     :return:
154     """
155     first_spike_list = []
156     for sample in data:
157       p.setup() # needed for PyNN
158
159       # the neurons providing the input signals are always the same,
    only their activity changes
160       input_neurons = p.Population(self.n_inputs, p.SpikeSourceArray, {
    'spike_times': sample})
161
162       # setup the output neuron population
163       output_neurons = p.Population(len(self.psp_thresholds), p.
    IF_curr_exp, cellparams=self.cell_params_out)
164       for i, neuron in enumerate(output_neurons):
165         if self.psp_thresholds[i] is not None:
166           neuron.set(v_thresh=self.psp_thresholds[i])
167
168       # weight list is based on the order of the incoming spikes
169       # need to go through the input spike trains and rank them
170       new_weights = []
171       self.weight_list.append(new_weights)
172
173       # setup the connections
174
175       # setup drift - here using STDP instead of SDSP because sPyNNaker
     does not support SDSP
176       timing_rule_drift = p.SpikePairRule(tau_plus=20.0, tau_minus
    =20.0)
177       weight_rule_drift = p.AdditiveWeightDependence(w_min=0.1, w_max
    =1.0, A_plus=0.02, A_minus=0.02)
178       drift_model = p.STDPMechanism(timing_dependence=timing_rule_drift
    , weight_dependence=weight_rule_drift)
179       network = p.Projection(input_neurons, output_neurons, p.
    FromListConnector(self.input_connection_list),
180                              synapse_dynamics=drift_model)
181
182       # add recorder for the spike times and membrane potentials
183       output_neurons.record(['spikes'])
184
185       # run simulation
186       p.run(self.t_sample)
187
188       # save output weights and traces
189       # traces & spike times first:
190       data_out = output_neurons.get_data().segments[0]
191       output_neurons.get_data(clear=True)
192
193       p.end() # needed for PyNN
194
195       first_spike_time = self.t_sample
196       first_spike_index = None
197       for i, time in enumerate(data_out):
198         if time <= first_spike_time:
```

```
            first_spike_time = time
            first_spike_index = i
        first_spike_list.append(first_spike_index)

    output = []
    for spike_time in first_spike_list:
        output.append(self.class_labels["{0}".format(spike_time)])

    return output


def merge_neurons(comparing_weights, current_sample_weights,
    n_previous_merges, previous_psp_threshold,
          current_psp_threshold):
    """

    :param comparing_weights:
    :param current_sample_weights:
    :param n_previous_merges:
    :param previous_psp_threshold:
    :param current_psp_threshold:
    :return:
    """
    new_weights = (current_sample_weights + comparing_weights *
      n_previous_merges) / n_previous_merges + 1
    new_threshold = (current_psp_threshold + previous_psp_threshold *
      n_previous_merges) / n_previous_merges + 1
    return new_weights, new_threshold


def find_max(data):
    """

    :param data:
    :return:
    """
    max_psp = -100.
    for point in data:
        if point < max_psp:
            max_psp = point
    return max_psp
```

Listing C.5: Implementation of the Dynamic Evolving SNN Classifier

C.6 GenericEncoder

```
class GenericEncoder():

    def __init__(self):
        pass

    def modify_encoder(self, **kwargs):
        raise NotImplementedError("Do not use GenericEncoder. No specific
        encoder defined.")

```

```
  def encode_spikes(self, **kwargs):
    raise NotImplementedError("Do not use GenericEncoder. No specific encoder defined.")
```

Listing C.6: Superclass of any Encoders implemented

C.7 TemporalDifferenceEncoder

```
import GenericEncoder # not really used at the moment but will be important when I introduce other encoders
from Utilities import load_csv_emotiv
import os
import re

def natural_key(string_):
  """See http://www.codinghorror.com/blog/archives/001018.html"""
  return [int(s) if s.isdigit() else s for s in re.split(r'(\d+)', string_)]

class TemporalContrast(GenericEncoder.GenericEncoder):
  def __init__(self, threshold=6., timestep=7.8125, encode_inhibitory=False):
    """
    Setup an Address Event Representation-style threshold based spike encoder.
    @param threshold: float threshold to spike at
    @param timestep: float time step between data points (1000/data collection rate per second)
    @param encode_inhibitory: boolean whether we want to encode inhibitory spikes as well as excitatory
    """
    # default is same default used in MATLAB version
    self.threshold = threshold
    # default is 128 Hz for Emotiv device
    self.timestep = timestep
    # do we encode inhibitory spikes?
    self.inhibitory = encode_inhibitory

  def modify_encoder(self, **kwargs):
    pass

  def encode_spikes(self, data_directory):
    """
    Wrapper to encode all data files in the given directory
    :param data_directory:
    :return:
    """
    spikes = []
    folder_list = next(os.walk(data_directory))[1]
    folder_list.sort(key=natural_key)
    for file_name in folder_list:
      data = load_csv_emotiv(data_directory + file_name)
      spikes.append(self._encode_spikes(data))
    return spikes

```

```
42   def _encode_spikes(self, data):
43     """
44     Actual method to encode spikes
45     :param data:
46     :return:
47     """
48     cols = len(data)
49     curr_time = 0.0
50     spiketimes = [[] for i in cols]
51     for col in data:
52       previous_value = 0
53       for row in col:
54         if row >= previous_value + self.threshold:
55           spiketimes[col].append(curr_time)
56         elif self.inhibitory: # we only need to check this if the prev
     is false
57           raise NotImplementedError("Inhibitory input spikes are not
     yet implemented.")
58         previous_value = row
59         curr_time += self.timestep
60     return spiketimes
```

Listing C.7: Implementation of a simplified Temporal Difference Encoder

C.8 Deprecated: NeuCubeStructure

```
1  from pyNN.space import BaseStructure
2  import random as rand
3  import numpy as np
4  import csv
5
6
7  class NeuCubeStructure(BaseStructure):
8
9    def __init__(self, filename='model_data/neuron_positions.txt',
10                    inhibitory_probability = 0.2):
11     self.filename = filename
12     self.inhibitory_probability = inhibitory_probability
13     self.excitatory_locations = []
14     self.inhibitory_locations = []
15     self.all_locations = []
16
17   def load_locations(self):
18     csv_file = open(self.filename, 'r')
19     reader = csv.reader(csv_file, delimiter=',')
20
21     # Read neuron positions from file
22     excitatory_list = []
23     inhibitory_list = []
24     for line in reader:
25       p = (float(line[0])/10., float(line[1])/10.,
26                    float(line[2])/10.)
27       if rand.random() < self.inhibitory_probability:
28         inhibitory_list.append(p)
29       else:
```

```
30          excitatory_list.append(p)
31
32      # convert to numpy array
33      self.excitatory_locations = np.array(excitatory_list)
34      self.inhibitory_locations = np.array(inhibitory_list)
35      # needs to be 3xn for the Population creation, is currently
36      # nx3 so transpose
37      self.excitatory_locations = self.excitatory_locations.T
38      self.inhibitory_locations = self.excitatory_locations.T
39
40      return self.excitatory_locations, self.inhibitory_locations
41
42  def get_inhibitory_locations(self):
43      return self.inhibitory_locations
44
45  def get_excitatory_locations(self):
46      return self.excitatory_locations
47
48  def generate_positions(self, num_neurons):
49      """
50      Generates positions for neuron population of num_neurons
51      based on the Talairach atlas.
52      Read in positions from file here and return positions_list
53      @param num_neurons:
54      @return:
55      """
56      csv_file = open(self.filename, 'r')
57      reader = csv.reader(csv_file, delimiter=',')
58
59      # Read neuron positions from file
60      position_list = []
61      for line in reader:
62          p = (float(line[0])/10., float(line[1])/10.,
63                          float(line[2])/10.)
64          position_list.append(p)
65
66      # convert to numpy array
67      positions = np.array(position_list)
68      # needs to be 3xn for the Population creation, is currently
69      # nx3 so transpose
70      positions = positions.T
71
72      return positions
```

Listing C.8: Deprecated method of implementing structure and distance for the reservoir, implemented with PyNN

Appendix D

Source and Version Control

As is best practice in software development, source and version control was implemented for all of the software development through the duration of this thesis. In this case, the distributed version control system Git was used, and mirrored in a private GitHub repository.[1]

As a part of the previously defined attempts to bring the development of the NeuCube framework as a whole under control, and to impose the design principles established in Chapter 6, version control systems and rules for all of the other modules of NeuCube have been implemented. This development is now handled through a set of private, centralised repositories[2] for each NeuCube module. These repositories provide a number of useful features for collaborative development, including:

1. Source control and histories (audit log);

This allows us to see what has changed and why, and provides a facility for code rollbacks should a bug or incorrect code be introduced. This also allows us to ensure accountability for the code that is introduced, and impose some requirements such as regression testing or linting checks before code is added to the public repository.

2. Centralised source storage;

The code is now centrally available to all approved users, at its current stage of development. This should help to mitigate the issue where a number of different people are working concurrently on the same version of the code in isolation, then futher building off that code, diverging what should be a collaborative project. Since the introduction of these centralised repositories, developers are able to collaboratively develop the same codebase and share their advances more easily, mitigating the aforementioned code divergence.

3. Release management tools;

[1] https://github.com/nmscott/NeuCube_PyNN
[2] https://github.com/KEDRI-AUT

Code can now be frozen as 'releases' at any stage, and released as a stable download which can be made publically available. In addition, the ability to create persistent Digital Object Identifier (DOI) archival numbers (identical to those used in journal papers) for these releases directly from the release management system has been incorporated.

4. Access control;

These repositories are private by default, and are only made accessible to selected users.[3] In addition, different access privileges can be issued to those users (read-only, write-with-permission, *etc.*) on an as-needed basis. Repositories can be made public (open access) if open source release of any of this code occurs in the future. Repositories can have their own access privileges applied. For example, users wishing to contribute to the PyNN repository are not able to do so at will – they must first develop their code in a 'fork' and then submit this changed code in a 'pull request', which the administrator of that repository must then review before combining the code. This process can be automated, and can incorporate minimum levels of regression testing or code formatting requirements.

5. Bug and Feature tracking;

Each repository now incorporates a comprehensive integrated bug reporting and feature request system. This allows users to report bugs or request new features, and have these assigned to developers and automatically tracked for progress.

6. User-editable Wikis;

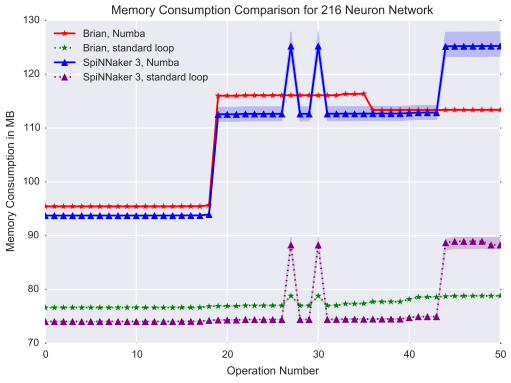
A wiki editable by all approved users of the repository is now linked to each project. We can use these to centralise important documents such as development or user manuals, or related publications in a readily updatable and easily distributable manner.

Unfortunately as this larger scale repository system was not implemented until late in this thesis, empirical measures to prove the efficacy of such systems on collaborative development in the NeuCube environment are not available. Heuristically, a distributed version control system has improved the development of the PyNN NeuCube version, particularly the ability to branch and merge code. Branching was used when developing for the early quirks of the SpiNNaker PyNN implementation, as this had some inconsistencies with the reference implementation of PyNN.

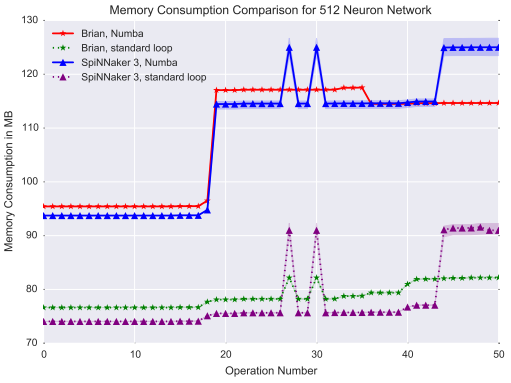
[3] Due to this access control restriction, the reader will likely find that they are unable to view the repositories linked here. Access can be requested from the author or KEDRI.

Appendix E

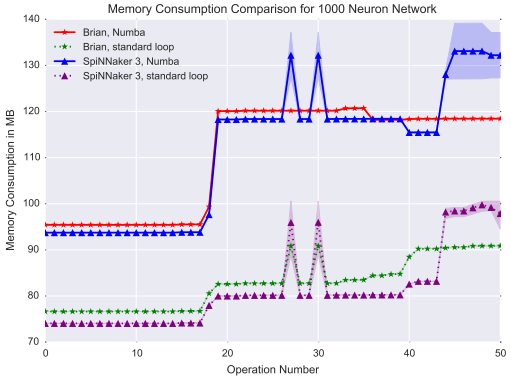
Memory Profiles of NeuCube Implementation in PyNN



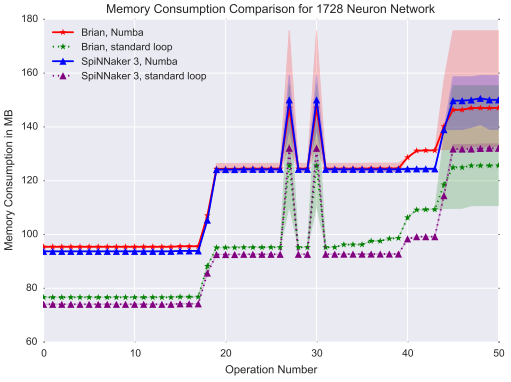
(A) Memory consumption for a 216 neuron NeuCube reservoir.



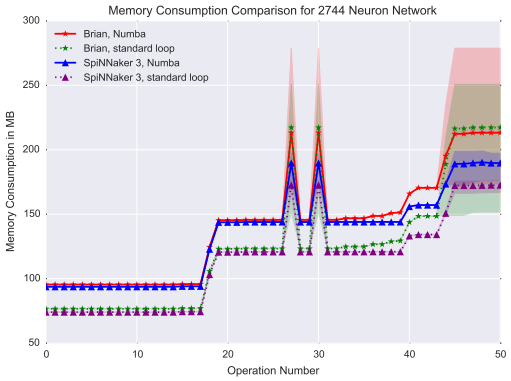
(B) Memory consumption for a 512 neuron NeuCube reservoir.



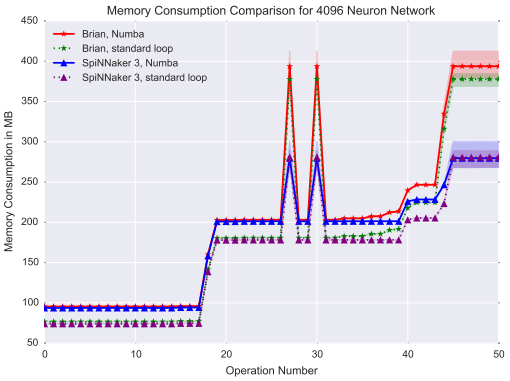
(C) Memory consumption for a 1,000 neuron NeuCube reservoir.



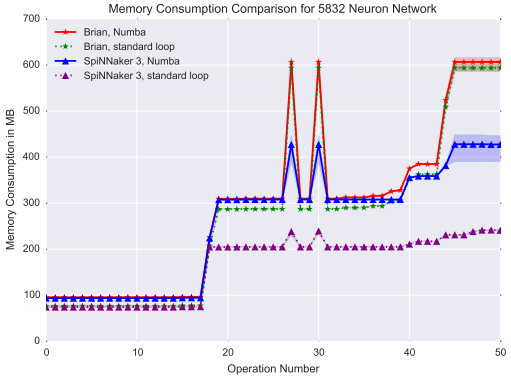
(D) Memory consumption for a 1,728 neuron NeuCube reservoir.



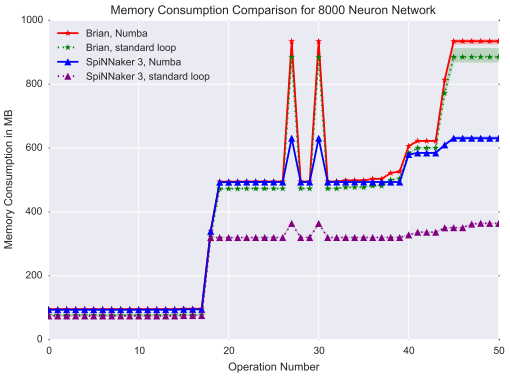
(e) Memory consumption for a 2,744 neuron NeuCube reservoir.



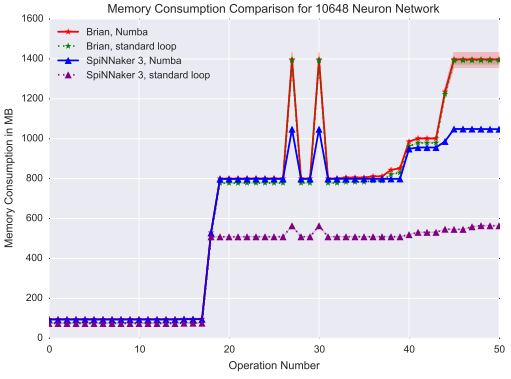
(f) Memory consumption for a 4,096 neuron NeuCube reservoir.



(g) Memory consumption for a 5,832 neuron NeuCube reservoir.



(h) Memory consumption for a 8,000 neuron NeuCube reservoir.



(i) Memory consumption for a 10,648 neuron NeuCube reservoir.

Figure E.1: Overall memory consumption for a simulation of the NeuCube reservoir on software and SpiNNaker simulation systems. Operation numbers represent method calls or line operations in the NeuCubeReservoir.py file. Memory consumption is generally lower overall when using a neuromorphic computing platform. Shading represents the 95% confidence interval for that value.

Appendix F

Basic Configuration file in JSON

```
{
  "location_prefix" : "model_data/",
  "data":
  {
    "data_directory" : "data/example_experiment/",
    "n_samples" : 30,
    "training_proportion" : 80.0
  },
  "reservoir":
  {
    "model_data_location" : "exp_1000.csv",
    "input_data_location" : "input_positions.txt",
    "excitatory_proportion" : 80.0,
    "p_connection" : 0.02,
    "cell_parameters": {"cm": 0.25, "i_offset": 0.0, "tau_m": 10.0, "tau_refrac": 2.0, "tau_syn_E": 3.0,
              "tau_syn_I": 3.0, "v_reset": -65.0, "v_rest": -65.0, "v_thresh": -50.0}
  },
  "run":
  {
    "sim_time" : 1000,
    "times_to_run" : 1,
    "plot_spikes" : true,
    "show_plots" : true
  }
}
```

Listing F.1: Example NeuCube configuration file in JSON format

Appendix G

Listing of Method for the Generation of Simulated Radioastromony Events

```
duration = 180; % This is the length of the observation
freq_n = 10; % Number of frequency bands you want to produce
instance = 1; % number of instances you want to have for each class.
max_sigma = 5; % number of maximum deviation of the pulse

r = 0 + 1.*randn(duration,freq_n,instance);
p = 0 + 1.*randn(duration,freq_n,instance);

for i=1:instance
  %For every instance generate a random number as the strength of the
  % pulse
  m = 1+ max_sigma*rand(1,1);

  %For every instance generate a random number as the width of the
  % pulse
  width = randi([2 5],1,1);

  %For every instance generate a random number as the interval between
  %the pulses
  interval = randi([200 500],1,1);

  %initialise a random value as a starting point of the signal for
  %that particular instance
  offset = randi([15 40],1,1);

  %iterate through all the frequency bands
  for j=1:freq_n
    offset = offset+round((j*j)/10);
    for k=offset:interval:duration-width
      for l=1:width
        p(k+l,j,i) = p(k+l,j,i) + m;
      end
    end
  end
end
```

```
36
37 %display the greymap of the first positive instance just to verify
38 imagesc(p(:,:,1)');
39 colormap(gray);1
40
41 eeg_data = cat(3, r, p);
42 class_label = [ones(1,instance) ones(1,instance)+1];
43
44 flat = reshape(eeg_data, [duration*freq_n instance*2]);
45 flat = flat';
46 flat = [1:(duration*freq_n)+1 ; flat];
47 csvwrite('ska-flat.csv',flat);
```

Listing G.1: Method for the generation of simulated radioastromony events. Implemented in MATLAB by R. Hartono and the author.