SpiNNaker

A Spiking Neural Network Architecture

Steve Furber and Petruţ Bogdan
(Editors)

now
the essence of knowledge

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

ISBN: 978-1-68083-652-3
E-ISBN: 978-1-68083-653-0
DOI: 10.1561/9781680836530

Suggested citation: Steve Furber and Petruţ Bogdan (eds.). (2020). *SpiNNaker – A Spiking Neural Network Architecture*. Boston–Delft: Now Publishers

Table of Contents

Preface ix

Acknowledgements xi

Glossary xii

Chapter 1 Origins 1

By Steve Furber

1.1 From Ada to Alan — Early Thoughts on Brains and Computers 1

1.1.1 Ada Lovelace 2

1.1.2 Alan Turing 3

1.2 Reinventing Neural Networks — Early Thoughts on the Machine 4

1.2.1 Mighty ARMs from Little Acorns Grow 4

1.2.2 Realising Our Potential 5

1.2.3 Reinventing Neural Networks 5

1.3 The Architecture Comes Together 6

1.3.1 The State of the Neuromorphic Art 6

1.3.2 What Could We Bring to Neuromorphics? 7

1.3.3 Multicast Packet-switched AER 8

1.3.4 Optimise, Optimise… 8

1.3.5 Flexibility to Cope with Uncertainty 9

1.3.6 Big Memories 10

1.3.7 Ready to Go 10

1.4 A Scalable Hardware Architecture for Neural Simulation 11

1.4.1 Introduction 11

1.4.2 Intellectual Property 11
1.4.3 Market Opportunity 11
1.4.4 System Organisation 12
1.4.5 Node Organisation 13
1.4.6 System Architecture Issues 15
1.4.7 Development Plan 15
1.5 Summary 16

Chapter 2 The SpiNNaker Chip 17
By Jim Garside and Luis A. Plana
2.1 Introduction 17
2.2 Architecture 18
2.2.1 An Overview 18
2.2.2 Processor Subsystem 27
2.2.3 Router 30
2.2.4 Interconnection Networks 36
2.2.5 The Rest of the Chip 39
2.3 Multiprocessor Support 41
2.4 Event-Driven Operation 42
2.5 Chip I/O 43
2.6 Monitoring 43
2.7 Chip Details 44
2.8 Design Critique 45
2.9 Summary 50

Chapter 3 Building SpiNNaker Machines 52
By Luis A. Plana, Steve Temple, Jonathan Heathcote, Dave Clark, Jeffrey Pepper, Jim Garside and Steve Furber
3.1 Putting Chips Together 53
3.1.1 SpiNN-3: Development Platform 54
3.1.2 SpiNN-5: Production Board 55
3.1.3 Nobody is Perfect: Testing and Blacklisting 60
3.2 Putting Boards Together 61
3.2.1 SpiNNaker Topology 62
3.2.2 spiNNlink: High-speed Serial Board-to-Board Interconnect 62
3.3 Putting Everything Together 67
3.3.1 *SpiNNaker1M* Assembly 67
3.3.2 *SpiNNaker1M* Interconnect 69
3.3.3 *SpiNNaker1M* Cabling 71

3.4 Using the Million-Core Machine: Tear it to Pieces 73
3.5 *SpiNNaker1M* in Action 74

Chapter 4 Stacks of Software Stacks 77

By Andrew Rowley, Oliver Rhodes, Petruț Bogdan, Christian Brenninkmeijer, Simon Davidson, Donal Fellows, Steve Furber, Andrew Gait, Michael Hopkins, David Lester, Mantas Mikaitis, Luis Plana, and Alan Stokes

4.1 Introduction 78
4.2 Making Use of the SpiNNaker Architecture 79
4.3 SpiNNaker Core Software 80
4.4 Booting a Million Core Machine 81
4.5 Previous Software Versions 84
4.6 Data Structures 85
4.6.1 SpiNNaker Machines 85
4.6.2 Graphs 86
4.7 The SpiNNTools Tool Chain 88
4.7.1 Setup 89
4.7.2 Graph Creation 89
4.7.3 Graph Execution 89
4.7.4 Return of Control/Extraction of Results 93
4.7.5 Resuming/Running Again 93
4.7.6 Closing 94
4.7.7 Algorithms and Execution 94
4.7.8 Data Recording and Extraction 95
4.7.9 Live Interaction 97
4.7.10 Dropped Packet Re-Injection 97
4.7.11 Network Traffic Visualisation 98
4.7.12 Performance and Power Measurements 99
4.8 Non-Neural Use Case: Conway's Game of Life 99
4.9 sPyNNaker – Software for Modelling Spiking Neural Networks 101
4.9.1 PyNN 103
4.9.2 sPyNNaker Implementation 104
4.9.3 Preprocessing 104
4.9.4 SpiNNaker Runtime Execution 107
4.9.5 Neural Modelling 118
4.9.6 Auxiliary Application Code 122
4.10 Software Engineering for Future Systems 124
4.11 Full Example Code Listing 125

Chapter 5 Applications — Doing Stuff on the Machine 127
By Petruț Bogdan, Robert James, Gabriel Fonseca Guerra, Garibaldi Pineda García and Basabdatta Sen-Bhattacharya
5.1 Robot Art Project 128
5.1.1 Building Brains with Nengo and Some Bits and Pieces 129
5.2 Computer Vision with Spiking Neurons 133
5.2.1 Feature Extraction 133
5.3 SpiNNak-Ear — On-line Sound Processing 137
5.3.1 Motivation for a Neuromorphic Implementation 138
5.3.2 The Early Auditory Pathway 138
5.3.3 Model Algorithm and Distribution 139
5.3.4 Results 140
5.3.5 Future Developments 142
5.4 Basal Ganglia Circuit Abstraction 143
5.5 Constraint Satisfaction 147
5.5.1 Defining the Problem 149
5.5.2 Results 152
5.5.3 Graph Colouring 153
5.5.4 Latin Squares 155
5.5.5 Ising Spin Systems 156
Chapter 6 From Activations to Spikes 160
By Francesco Galluppi, Teresa Serrano Gotarredona, Qian Liu and Evangelos Stromatias
6.1 Classical Models 161
6.2 Symbol Card Recognition System with Spiking ConvNets 162
6.2.1 Spiking ConvNet on SpiNNaker 165
6.2.2 Results 166
6.3 Handwritten Digit Recognition with Spiking DBNs 169
6.3.1 Results 173
6.4 Spiking Deep Neural Networks 178
6.4.1 Related Work 179
6.4.2 Siegert: Modelling the Response Function 181
6.4.3 Generalised Off-line SNN Training 191
6.4.4 Results 196
6.4.5 Summary 203
Chapter 7 Learning in Neural Networks 205
By Petruț Bogdan, Garibaldi Pineda García, Michael Hopkins, Edward Jones, James Knight and Adam Perrett
7.1 Sizing Up the (Biological) Competition 206
7.2 Spike-Timing-Dependent Plasticity 206

7.2.1 Experimental Evidence for Spike-Timing-Dependent Plasticity ... 207
7.2.2 Related Work ... 213
7.2.3 Implementation ... 215
7.2.4 Inhibitory Plasticity in Cortical Networks ... 221
7.2.5 The Effect of Weight Dependencies ... 222
7.3 Voltage-Dependent Weight Update ... 224
7.3.1 Results ... 227
7.4 Neuromodulated STDP ... 231
7.4.1 Eligibility Traces/Synapse Tagging ... 232
7.4.2 Credit Assignment ... 234
7.5 Structural Plasticity ... 236
7.5.1 Topographic Map Formation ... 238
7.5.2 Stable Mappings Arise from Lateral Inhibition ... 243
7.5.3 MNIST Classification in the Absence of Weight Changes ... 246
7.5.4 Visualisation, Visualisation, Visualisation ... 248
7.5.5 Rewiring for Motion Detection ... 249
7.6 Neuroevolution ... 255
7.6.1 Pac-Man on SpiNNaker ... 257
7.6.2 Further Exploration of NEAT ... 257
7.6.3 An Evolutionary Optimisation Framework for SpiNNaker ... 257
7.6.4 Methods ... 258
7.6.5 Results ... 259
7.6.6 Future Work ... 259

Chapter 8 Creating the Future 262

By Dave Clark, Andreas Dixius, Steve Furber, Jim Garside, Michael Hopkins, Sebastian Höppner, Dongwei Hu, Florian Kelber, Gengting Liu, Christian Mayr, Mantas Mikaitis, Felix Neumärker, Johannes Partzsch, Stefan Schiefer, Stefan Scholze, Delong Shang and Marco Stolba

8.1 Survey of Currently Available Accelerators ... 262
8.2 SpiNNaker2 ... 264
8.2.1 Lessons from SpiNNaker1 ... 264
8.2.2 Scaling Performance and Efficiency ... 265
8.3 SpiNNaker2 Chip Architecture ... 266
8.4 SpiNNaker2 Packet Router ... 268
8.5 The Processing Element (PE) ... 270
8.5.1 PE Components ... 271
8.5.2 PE Implementation Strategy and Power Management ... 277
8.6 Summary ... 279

References 280
Index 306
About the Editors 318
Contributing Authors 319

Preface

In the quarter of a century following the invention of the microprocessor, computers had become a thousand times more powerful and had more than a thousand times more memory, yet they still struggled to perform certain classes of task that humans – even very young humans – find straightforward, such as recognising a mother's face or identifying a three-dimensional object at any range or orientation. How is it that biological brains, built from pretty slow basic technology, can still outperform machines, at least on certain tasks, that are built using technologies a million times faster?

Although we know aspects of the answer to the question about the brain's capabilities, such as its exploitation of massive parallelism, we are still a long way from understanding the full answer. The principles of operation of the brain as an information processor remain as one of the great frontiers of science. Neuroscientists, working from the bottom up, know a great deal about the components – neurons – from which the brain is constructed and can observe activities of increasing numbers of them. Psychologists and brain imagers work from the top down and can see activity moving across the coarse structures of the brain. But understanding information processing in the brain, in the same way that we understand information processing in a computer, requires that we look at intermediate levels that are not observable with either bottom-up or top-down approaches, and the only 'instrument' we have to test hypotheses about how information is coded in spatio-temporal patterns of neural 'spikes', how these are stored and then recovered, and how information is passed from one region of the brain to another, is the computer model. From this line of thinking, the SpiNNaker project emerged. SpiNNaker is a contraction of *Spiking Neural Network Architecture*, an approach to building a machine that is based to some degree on what is understood about the principles of operation of the brain. These principles include the use of spikes – electro-chemical impulses that are pure asynchronous events – as the primary mode of real-time communication, and the very high degree of connectivity found in the biological brain where each neuron typically connects to many thousands of other neurons.

The research was ultimately configured to address two high-level questions:

- Can the enormous computational power available today be used to accelerate our understanding of the principles of operation of information processing in the brain?
- Can we use our growing understanding of the brain to build better, more efficient, fault-tolerant computers?

This book is the story of the first 20 years of this research programme, an outcome of which is the world's largest neuromorphic computing platform ultimately incorporating a million processor cores, capable of modelling spiking neural networks of the scale of a mouse brain in biological real time. The mouse brain is around a thousand times smaller than, but in some senses otherwise very similar to, the human brain. So there is still a long way to go to deliver a real-time model of a full human brain, but SpiNNaker can support sophisticated and biologically-realistic models of substantial brain subsystems, albeit with the emphasis on the details of the network topology rather than the internal complexities of the individual neurons.

Steve Furber, 12 July 2019

Acknowledgements

Many staff, students and collaborators have contributed to this story over the last two decades; quite a number of them have contributed to the writing of this book, though there are many more whose names do not appear directly here, but whose contributions are nevertheless an integral part of the SpiNNaker story. Their contributions are all gratefully acknowledged.

Funding Acknowledgements

The design and construction of the SpiNNaker machine was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/D07908X/1 and EP/G015740/1, in collaboration with the universities of Southampton, Cambridge and Sheffield and with industry partners ARM Ltd, Silistix Ltd and Thales. Ongoing development of the software is supported by the EU ICT Flagship Human Brain Project (FP7-604102, H2020 720270 and H2020 785907), in collaboration with many university and industry partners across the EU and beyond, and exploration of the capabilities of the machine was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement 320689. The support of these funding agencies is gratefully acknowledged.

Glossary

A

ABB - *Adaptive Body Biasing* - a technique used on a CMOS chip to reduce the impact of manufacturing variability on the performance of the chip. 277, 279

AD - *Absolute Deviation* - the deviation of the mean receptive field from its ideal location in a topographic map. 240, 241, 243, 244

ADPLL - *All Digital Phase-Locked Loop* - a digital circuit that controls the frequency and phase of one signal to match the frequency and phase of a reference signal. 273

AER - *Address Event Representation* - a mechanism for encoding spike events from an SNN as a stream of numbers or 'addresses', where each address corresponds to a spike from a particular neuron. 6–10, 19, 22, 55, 107, 164, 166

AGI - *Artificial General Intelligence* - AI approaching human-like general-purpose capabilities. 128

AHB - *AMBA High-Performance Bus* - a multi-master bus protocol introduced in AMBA version 2. A simple transaction on the AHB consists of an address phase and a subsequent data phase. 27, 28, 34, 39, 275

AI - *Artificial Intelligence* - a term applied broadly to machine learning systems that display some specific human-like capability, such as the ability to play chess or to recognise cats in an image. 128, 155, 156, 160, 161, 163, 179, 203, 263

AN - *Auditory Nerve* - a bundle of axons representing the output of the cochlea. 139–142

ANN - *Artificial Neural Network* - a conventional neural network where neurons produce continuously varying outputs rather than spikes, usually trained using error back-propagation. 161, 178–181, 188, 190, 191, 193–196, 199–201, 203, 204, 246, 255, 257, 260–262, 264

API - *Application Programming Interface* - a software interface that abstracts hardware details into useful software functions. 104, 106, 118, 122

ARM - *Acorn RISC Machine* - although this expansion is now deprecated, the term is used to describe the company ARM Ltd or the range of microprocessor architectures that they design and that are widely used in mobile phones and many other computer systems, including SpiNNaker. 4, 5, 7, 9, 10, 14, 16, 18, 19, 25, 27, 28, 30, 34, 39, 41, 43, 44, 48, 49, 51, 54, 55, 57, 67, 68, 73, 76, 80, 120, 127, 172, 176, 207, 216, 217, 263, 266, 270, 272, 275–277

ASIC - *Application-Specific Integrated Circuit* - a microchip designed for a specific purpose. 18–21, 25, 27, 43–45, 264

AXI - *Advanced eXtensible Interface* - a third-generation AMBA interface, targeted at high-performance systems, that allows the issuing of multiple outstanding commands with out-of-order responses. 28, 39

B

BG - *Basal Ganglia* - a brain region responsible for action selection, among others. 128, 132, 143–146

blinkenlight - a neologism for diagnostic lights on the front panel of electronic equipment. 43, 80

BMP - *Board Management Processor* - an ARM-based microcontroller used to control the operation of SpiNNaker boards; the BMP is in charge of powering up and down the SpiNNaker chips, configuring board FPGAs, controlling cooling fan speed, keeping track of board operating temperatures and taking appropriate action in case of overheating. 57, 59–61, 66–68, 74

C

CAM - *Content-Addressable Memory* - a form of computer memory that allows look-up by data content rather than by address. 22, 24

CAN - *Controller Area Network* - a computer network standard with well-defined real-time capabilities, used in automotive and other real-time applications. 68

CMOS - *Complementary Metal Oxide Semiconductor* - the semiconductor technology used to build most current mainstream microchips such as microprocessors and memory, comprising both n-type and p-type field-effect transistors (FETs). 16, 45, 51, 277

ConvNet - *Convolutional Neural Network* - a form of ANN used for image classification. 161–166, 169, 196, 197, 201, 264

CPU - *Central Processing Unit* - a hardware component responsible for basic arithmetic, logic, control and I/O. 88, 127, 141, 142, 217

CRC - *Cyclic Redundancy Check* - an algorithm for detecting errors in stored or communicated data. 30, 46, 64–66

CSP - *Constraint Satisfaction Problem* - a class of mathematical problems that need to satisfy a number of limitations. 148–154, 156, 158

D

DBN - *Deep Belief Network* - a form of DNN comprising a stack of restricted Boltzmann machines. 160, 169–178

DC - *Direct Current* - electric current flowing in one direction only. 56

DDR - *Double Data Rate* - a form of computer memory where data is delivered on both the rising and the falling edges of the clock. 43

DfT - *Design for Test* - a systematic approach to microchip design that takes the testability of the logic into account throughout the design process. 267

DLL - *Delay-Locked Loop* - a mechanism that adjusts the relative phases of different clock signals. 54

DMA - *Direct Memory Access* - a hardware mechanism for copying blocks of memory. 10, 25, 27–30, 39, 42, 46, 48, 51, 81, 107, 108, 112, 114, 116–118, 123, 213, 215, 267, 272

DNN - *Deep Neural Network* - an ANN with many layers of neurons. 161, 200, 263, 275, 276

DRNL - *Dual Resonance Non-Linear* - a filter bank modelling cochlear behaviour. 139, 140

DSI - *Direction Selectivity Index* - a metric describing the selectivity of a neuron's response to motion in particular direction. 251–254

DTCM - *Data Tightly-Coupled Memory* - an ARM name for a scratchpad memory holding program data. 27, 29, 82, 88, 106–108, 112–114, 122, 123, 165, 166, 214, 217

DVFS - *Dynamic Voltage and Frequency Scaling* - a technique to optimise the power efficiency of a circuit by adjusting its supply voltage and operating frequency to match the current workload. 279

DVS - *Dynamic Vision Sensor* - a camera that encodes changes in pixel luminance as asynchronous spike events. 55, 57, 164, 166, 170, 171

E

EA - *Evolutionary Algorithm* - a computer algorithm that optimises parameters using an approach more-or-less similar to biological evolution. 255–257, 260, 261

EoP - *End-of-Packet* - a special marker used to indicate the end of a sequentially-transmitted packet. 35–38

EPSRC - *The UK Engineering and Physical Sciences Research Council* - the UK's research funding body for engineering and the physical sciences. xii, 5, 16

ES - *Evolutionary Strategies* - a class of optimisation methods. 260

F

FIFO - *First In First Out* - a form of queue where outputs emerge in the order that they were input. 65, 267

FIQ - *Fast Interrupt Request* - a name used by ARM for an input signal to a microprocessor that interrupts the processor, at a higher priority than IRQ. 30, 81, 108, 112, 118

Flash - *Flash Memory* - solid-state, non-volatile data storage that can be electrically erased and reprogrammed. 55

flit - *flow control unit* - a link-level atomic piece that forms a network packet or stream. 36–38

FPGA - *Field-Programmable Gate Array* - a microchip that has logic that can be configured to perform an arbitrary function. 55, 57–60, 62, 63, 65, 66, 74, 178

FPU - *Floating Point Unit* - a hardware unit in a microprocessor that can handle operations on floating-point numbers. 172, 270

G

GA - *Genetic Algorithm* - a form of EA where the design of the system to be optimised is expressed in the form of a genetic code. 256, 258–260

GALS - *Globally Asynchronous Locally Synchronous* - a systematic approach to microchip design where modules are internally driven by the same clock but different modules have different clock frequencies and/or phases. 267

GAN - *Generative Adversarial Network* - a machine learning algorithm where two neural networks compete with each other, generating new data with statistics similar to the training data. 162

GNU - *GNU's Not Unix* - an extensive collection of free computer software. 80

GPU - *Graphics Processing Unit* - a hardware accelerator originally for graphics operations, but increasingly also used to accelerate other numerical algorithms including those required for ANNs. 263

GUI - *Graphical User Interface* - a mechanism to enable users to interact with a computer through a structured array of information on a screen, usually in the form of multiple windows where each window displays the state of one application. 72

H

HBP - *Human Brain Project* - the European Commission's 10-year Flagship project to advance the use of computer technology in brain research, in which SpiNNaker plays a role. 76, 103, 207, 219

HPC - *High-Performance Computer* - a massively-parallel computer with fast interprocessor communication delivering the maximum computational throughput possible on a multi-million dollar budget. 9, 207

HSSL - *High-Speed Serial Link* - a technique for moving data at high-speed between microchips or computer subsystems. 63–65

I

IHC - *Inner Hair Cell* - motion sensitive cell located in the cochlea. 138–141

IP - *Internet Protocol* - standards used to regularise communications on the internet. 54, 55, 68, 80, 81, 88

ISI - *Inter-spike interval* - the time between two consecutive spikes in an SNN. 216

ITCM - *Instruction Tightly-Coupled Memory* - an ARM name for a scratchpad memory holding executable code. 27, 29, 80–82, 106

J

JTAG - *Joint Test Action Group* - a standard for testing microchip I/O and internal functions. 43, 59

L

L2L - *Learning-to-Learn* - a learning framework involving evolving networks capable of performing well for a family of tasks. 261

LCD - *Liquid-Crystal Display* - a form of flat-panel display that exploits the ability of liquid crystal to modulate light passing through them. 68

LED - *Light-Emitting Diode* - a semiconductor device that emits light when a current flows through it. 55, 59, 72

LIF - *Leaky Integrate and Fire* - a point-neuron model that accumulates inputs until a threshold is reached, whereupon it emits a spike and resets itself; the accumulation is 'leaky', so it decays over time in the absence of further input. 106, 118, 120, 121, 149, 170, 171, 178, 180–185, 187–193, 195–199, 201, 203, 204, 214, 223, 225, 226, 235, 249

LPDDR4 - *Low-Power Double Data Rate version 4* - a particular SDRAM standard. 266, 267

LSTM - *Long Short-Term Memory* - a particular mechanism for introducing state into an ANN. 162

LTD - *Long-Term Depression* - a synaptic plasticity mechanism that decreases synaptic strength over a long time period. 235

LTP - *Long-Term Potentiation* - a synaptic plasticity mechanism that increases synaptic strength over a long time period. 228, 235

LUT - *Look-up table* - a table containing pre-computed values that would otherwise be expensive to compute at the time of use. 274

M

MAC - *Media Access Control* - the low-level mechanism used to allow computers to connect to the internet (see also: MAc - *Multiply-Accumulate*). 43, 54, 55, 68

MAc - *Multiply-Accumulate* - an arithmetic unit used to compute an inner product (see also: MAC - *Media Access Control*). 275, 276

MC - *Multicast* - a communication protocol used by SpiNNaker to send a single spike packet to many, though not all possible, destinations. 31–33, 141, 172, 270

MII - *Media-Independent Interface* - standard interface that connects any type of PHY to any MAC independently of the network signal transmission media. 43

ML - *Machine Learning* - a field of study concerned with developing methods and algorithms such that computers perform specific tasks without explicit instructions. 162, 272, 275, 276

MNIST - *Modified NIST* - data set of images containing handwritten digits. 134, 170–172, 174, 176, 177, 197, 198, 238, 247, 258, 259, 261

N

NEAT - *NeuroEvolution of Augmenting Topologies* - an algorithm for evolving neural network topologies. 257

Nengo - *Neural ENGineering Objects* - a framework for SNNs grounded in control theory, from the University of Waterloo, Canada. 130, 131

NMDA - *N-Methyl-D-Aspartic Acid* - a particular neurotransmitter characterising a class of synapse. 228–230

NN - *Nearest Neighbour* - a communication protocol used by SpiNNaker to send a packet to an adjacent chip. 31, 33, 34, 38, 48, 82, 83, 270

NoC - *Network-on-Chip* - a regular mechanism for interconnecting diverse functional units on a microchip. 14, 15, 38, 39, 41, 44, 48, 51, 54, 266–268, 270–272, 275, 276

NP - *Non-deterministic Polynomial time* - a measure of the computational complexity of an algorithm. 148, 152–155, 158

NRZ - *Non-Return-to-Zero* - a signalling system where either transition – low to high or high to low – conveys the same information. 36, 38

NSP - *Noisy Softplus* - an activation function designed to closely simulate the firing activity of simple spiking neurons. 178, 180, 181, 189–197, 199–201, 203

P

P - *Polynomial time* - a measure of the computational complexity of an algorithm. 148, 153, 154

P2P - *Point-to-Point* - a communication protocol used by SpiNNaker to send a packet to a specific destination. 31, 40, 49, 83

PAF - *Parametric Activation Function* - a class of activation functions in the context of ANNs. 178, 191, 194–197, 201, 203

PC - *Personal Computer* - a computer designed for use by a single user, often identified with the IBM PC standard. 15, 129

PCB - *Printed Circuit Board* - a multilayer board, usually made of fibreglass, with copper interconnect patterns formed by a printing and etching process, used to mount and connect multiple microchips and other electronic components. 12, 36, 43, 49, 56, 58

PE - *Processing Element* - on SpiNNaker-2, a basic processing unit comprising an ARM Cortex M4F processor, memories and other local functions; see QPE. 265–268, 270–272, 275, 277–279

PHY - *Physical Layer Device* - connects a link layer device (MAC) to a physical medium such as a copper cable. 43, 267

PLL - *Phase-Locked Loop* - a control system that generates an output signal whose phase is related to the phase of an input signal. A simple implementation consists of a variable frequency oscillator and a phase detector in a feedback loop. 39, 279

POST - *Power-On Self-Test* - a system whereby a computer or device can check its own functionality every time it is switched on. 60

PRNG - *Pseudo-Random Number Generator* - a software or hardware algorithm for generating pseudo-random numbers. 273

PSP - *Post-Synaptic Potential* - the effect over time of an incoming spike on the membrane potential of the post-synaptic neuron. 150, 179

PVT - *Process, Voltage and Temperature* - the manufacturing and operational factors that affect the performance of a CMOS circuit. 277–279

Q

QPE - *Quad Processing Element* - a SpiNNaker-2 unit comprising a NoC router and four PEs; the SpiNNaker-2 chip largely comprises a tiled array of QPEs, each of which comprises a synchronous unit in a GALS organisation. 266–268, 270, 279

R

RAM - *Random-Access Memory* - computer memory where the contents can be read and written in any order. 8, 10, 14, 19, 25–27, 31, 34, 44, 46, 48, 49, 51, 147

RBM - *Restricted Boltzmann Machine* - a generative stochastic ANN where the neurons form a bipartite graph. 161, 162, 170, 189

ReLU - *Rectified Linear Unit* - a very popular activation function in the context of ANNs. 181, 189, 195–201, 203, 276

RISC - *Reduced Instruction Set Computer* - an architectural innovation emerging from work at UC Berkeley around 1980 that emphasizes simplicity and regularity in ISA design for microprocessors; contrast with CISC. 4

RMSE - *Root Mean Squared Error* - a widely used measure for the accuracy of the fit of an equation to a set of data. 247, 252

RNN - *Recurrent Neural Network* - a neural network where information flow is not simply unidirectional. 162

ROM - *Read-Only Memory* - a random access memory whose contents are fixed. 14, 26, 27, 29, 39, 40, 43, 54, 60, 67, 81

RTZ - *Return-To-Zero* - a signalling system where the high or low level on a wire conveys information, such as 1 and 0, respectively. 36

S

SARK - *SpiNNaker Application Runtime Kernel* - software written to allow access to the features of the SpiNNaker core and chip. 80, 81, 84, 105, 107

SATA - *Serial AT Attachment* - a particular HSSL standard for communication protocol and wiring; SpiNNaker uses the wiring standard, but not the protocol, for board-to-board connections. 57–60, 66–68, 73, 79

SCAMP - *SpiNNaker Control And Monitor Program* - software written to allow one of the cores to operate as a monitor processor through which the chip can be controlled. 80, 82–85, 95, 96

SDP - *SpiNNaker Datagram Protocol* - communication protocol employed on the SpiNNaker communication fabric. 80, 81, 95, 96

SDRAM - *Synchronous Dynamic Random-Access Memory* - a form of computer memory with high density and performance. 10–16, 19, 24, 27–30, 38–40, 43–46, 48, 51, 54, 60, 61, 79, 80, 85, 88–96, 100, 101, 106–108, 112–114, 118, 122, 123, 139, 165, 214, 265, 266, 272

SerDes - *Serialiser/Deserialiser* - an interface that converts parallel data to serial data and *vice versa* for high-speed inter-chip communication. 266

SNN - *Spiking Neural Network* - a neural network where communication between neurons is in the form of asynchronous impulses, or 'spikes', where information is conveyed only in the timing of the spikes. 77, 102–105, 107, 110, 116, 117, 119, 122, 127, 128, 136, 138, 149–158, 161, 170, 171, 176–181, 188, 189, 191, 192, 195–197, 199–204, 206, 207, 231, 250, 255, 257–262, 264, 268

SNr - *substantia nigra pars reticulata* - the output structure of the basal ganglia. 144, 145

SoC - *System-on-Chip* - a microchip that incorporates most of the required system functions, usually including one or more microprocessor cores, memories, on-chip buses or NoCs, specialized interfaces, etc. 12, 15, 16, 265

Softplus - an activation function in the context of ANNs. 183, 195, 197, 199–203

Spalloc - *SpiNNaker machine partitioning and allocation server* - the SpiNNaker job submission system that allocates a subset of the machine to individual user jobs. 74, 259

SPI - *Serial Peripheral Interface* - a synchronous serial communication interface specification used mostly in embedded systems. 43, 66

SpiN1API - *SpiNNaker1 API* - the SpiNNaker1 set of low-level, on-chip libraries implementing its event-based operating system. 81, 84, 105, 107, 108

SpiNNaker - *Spiking Neural Network Architecture* - a many-core neuromorphic computing platform. 1, 4, 8–10, 16–20, 24, 27, 30, 31, 36, 38–45, 47, 48, 50–69, 71–80, 82, 84–89, 95–99, 101, 103–108, 110, 116, 118, 120, 122, 127–130, 133, 138–147, 149, 152, 163–172, 176–178, 203, 205, 207, 210, 213–217, 220, 222, 227, 229, 233, 234, 236–238, 241, 248, 250, 251, 256–262, 264–266, 268–270, 272–274, 279

SpiNNer - SpiNNaker machine cabling planning, installation and verification software. 72, 73

spiNNlink - *SpiNNaker board-to-board link* - a High-Speed Serial Link (HSSL) used for SpiNNaker board-to-board interconnect. Implemented using FPGAs. 63–66, 73, 79

SpiNNTools - *SpiNNaker Tools* - the SpiNNaker software toolchain allowing execution of arbitrary problems defined in the form of a graph. 77, 78, 86, 88–90, 93, 94, 97, 101, 106, 111

sPyNNaker - *SpiNNaker backend for PyNN* - the SpiNNaker software simulator implementing PyNN constructs. 77, 103–107, 109, 111, 112, 116–118, 120–122, 146, 250

SRAM - *Static Random-Access Memory* - a form of computer memory that holds data in a static form, that is, as long as the memory is powered. 10, 15, 19, 25–27, 29, 30, 39, 40, 46, 48, 270, 272, 275–277

STDP - *Spike-timing-dependent plasticity* - a learning rule in SNNs relying on the relative difference between pre- and post-synaptic spikes when computing weight updates. 205–220, 222, 224–227, 231–233, 235–237, 239, 240, 247, 254, 274, 275

SWTA - *Soft Winner-Takes-All* - a neural mechanism whereby the most stimulated neuron in a group suppresses activity on other neurons in the group to a degree. 227, 228

T

TCAM - *Ternary Content-Addressable Memory* - a CAM where some of the bits in the content data that are being looked-up are ignored. 8–10, 32, 34, 35, 47, 49, 270

TCM - *Tightly-Coupled Memory* - an ARM name for a scratchpad memory that runs at full processor speed. 27, 28, 48

TM - *Tympanic Membrane* - the membrane separating the outer and middle ear. 138, 139

U

UDP - *User Datagram Protocol* - internet communication protocol. 80–82, 97

V

VIC - *Vectored Interrupt Controller* - a device that is used to combine several sources of interrupt onto one or more CPU lines, while allowing priority levels to be assigned to its interrupt outputs. 49

VLSI - *Very Large Scale Integration* - microchip technology whereby many transistors can be 'printed' on a single chip. 5, 6

W

WTA - *Winner-Takes-All* - a neural mechanism whereby the most stimulated neuron in a group suppresses activity on other neurons in the group completely. 151, 250, 254

Z

ZIF - *Zero-Insertion-Force* - a socket for a microchip that allows easy insertion and removal of the microchip. 54

DOI: 10.1561/9781680836530.ch1

Chapter 1

Origins

By Steve Furber

> *I have my hopes, and very distinct ones too, of one day getting cerebral phenomena such that I can put them into mathematical equations – in short, a law or laws for the mutual actions of the molecules of brain … I hope to bequeath to the generations a calculus of the nervous system.*
>
> — Ada Lovelace

The Spiking Neural Network Architecture (SpiNNaker) project has as its aim the design and construction of a massively parallel computer to support the modelling of large-scale systems of spiking neural networks in biological real time. The objectives of this research are two fold: firstly, to build a machine that can contribute to progress towards the scientific Grand Challenge of understanding the principles underpinning information processing in the brain; and secondly, to use what we do understand about the brain to help build better computers.

1.1 From Ada to Alan – Early Thoughts on Brains and Computers

The brain remains as one of the great frontiers of science – how does this organ upon which we all so critically depend do its job? We know a great deal about the low-level details of neurons and synapses, glial cells and mitochondria, and we can use brain imaging machines to see how activity moves around in the brain in response to external stimuli. But all of the interesting information processing takes place at intermediate scales reachable neither by bottom-up neuroscience nor by top-down brain imaging. The only tools available at these intermediate levels are those based on computer modelling, where we can test hypotheses about fundamental questions such as how does the brain learn and store new information, and how is what we see with our eyes represented in spatio-temporal patterns of spikes within our brains.

Interest in the brain is not new, of course. It took some time to determine that our central control system was based in our head, not in our heart, and even more time to understand the neuronal basis of this control [204]. But even before we achieved this basic level of understanding, there was speculation about what might be happening, and here we look at just two characters in this long story of working towards an understanding of how we operate as an allegedly intelligent species.

1.1.1 Ada Lovelace

Ada Lovelace (1815–1852) is known in computing circles primarily for her work as an assistant to Charles Babbage, who designed and tried to build mechanical computing engines in early Victorian times. She was the only legitimate child of Lord Byron and proposed an algorithm for Babbage's Analytical Engine (which was never built) that is viewed by many as the first computer program. She wrote copious notes about her work with Babbage and her thoughts generally, among which is the quote at the head of this chapter on her thoughts about creating a mathematical theory of the brain's operation.

Figure 1.1. Ada Lovelace.

Sadly, Ada never got to deliver on this ambition. She lived before Ramón y Cajal's revelations of the details of the neuron, but even today, with all the detailed knowledge gleaned in the interim years, her agenda would be considered highly ambitious!

1.1.2 Alan Turing

Alan Turing (1912–1954) is, perhaps, the highest profile figure in the history of computer science. In his seminal 1936 paper 'On Computable Numbers, with an Application to the Entschedungsproblem' [257], he proposed a 'universal computing machine' that could compute anything that is in principle computable. This universality is the basis for the modern programmable computer and is the basis of the ability of such machines to turn their hands to extremely diverse uses. Of course, no finite machine is truly universal, but that is not our concern here.

In September 1948, Turing moved to the University of Manchester, where the first machine to implement stored-program operation had run its first program on 21 June that year. Note that Turing did not contribute directly to the design or construction of the Manchester 'Baby' machine; Freddie Williams and Tom Kilburn had led that activity and achieved successful operation before Turing arrived in Manchester.

Turing performed various duties while in Manchester, and in 1950, he wrote his seminal paper 'On Computing Machinery and Intelligence' [256]. In this paper, he begins with the words 'I propose to consider the question "can machines think?" '. He then goes on to turn this around into what he calls 'The imitation game', but

Figure 1.2. Alan Turing (aged 16).

which subsequent generations know simply as the Turing test for human-like artificial intelligence.

Turing reckoned that all that a computer would need to pass his test, compared with the Manchester Baby machine, was more memory, about a gigabyte (a billion bytes) should be enough. (Baby had 128 bytes of memory and could execute some 700 instructions per second.) He thought that by the turn of the 21st century computers might have that much memory.

Indeed, by the turn of the 21st century a typical desktop computer would have about a gigabyte of memory, and it would be a million times more powerful than the Baby, but it would not pass Turing's test. This would have surprised Turing.

Why has human-like artificial intelligence proved so much harder than Turing, and many others since him, predicted? Perhaps it is because we still do not understand natural intelligence, so we do not know exactly what it is we are trying to reproduce in our machines. Natural intelligence is the product of the brain.

1.2 Reinventing Neural Networks – Early Thoughts on the Machine

1.2.1 Mighty ARMs from Little Acorns Grow

The origins of the SpiNNaker project can be traced back to 1998, although the motivation goes back even further. I (Steve Furber) spent the 1980s working for Acorn Computers Limited, a start-up based in Cambridge, UK, that rose to prominence as a result of securing the contract to develop the BBC Microcomputer, which was launched in January 1982. The BBC Micro, or simply 'Beeb', rode the wave of interest in personal computing that swept the world in the late 1970s and early 1980s. Acorn wished to build on this success and, through a strange sequence of events, ended up designing its own microprocessor based on principles emerging from US academia that went under the heading of the Reduced Instruction Set Computer (RISC). That microprocessor was the Acorn RISC Machine (ARM), over 120 billion descendants of which now power much of the world's mobile and embedded computing infrastructure.

The first ARM processor ran its first program on 26 April 1985; 13 years later, in 1998, processors were perhaps a hundred times faster, yet they still struggled to do some of the things that humans – even very young humans – find easy, such as recognising a human face or picking up a toy. I began to think about what it was that is different between how computers work and how brains work.

1.2.2 Realising Our Potential

The first concrete step towards addressing this question came when we had the opportunity to bid for a modest grant from EPSRC. EPSRC is the UK's Engineering and Physical Sciences Research Council, which funds UK academic research in the areas suggested in its name, including computing. By virtue of past support from ARM Ltd (the processor was now in the hands of a company that had inherited its name!), I was eligible to apply for a grant under a scheme called ROPA – Realising Our Potential Award. One of the requirements of this scheme was that the proposed research should be in a different direction from previous work, which in my case had focused on the Amulet series of asynchronous (clockless) implementations of the ARM processor. In an attempt to begin to think about how electronic systems might work a little more like the brain, I came up with a proposal entitled 'Efficient Very Large Scale Integration (VLSI) Architectures for Inexact Associative Memories'. The summary on the grant application read: 'Sequential processors are remarkably good at some tasks and remarkably poor at others. Many tasks in the second category can be described under the general heading of association – recognising similarities. Although alternative architectures such as neural networks have been proposed as efficient ways to implement inexact association, they have frequently not led to designs that match current VLSI technology well. The proposed research will investigate alternative ways to implement inexact association with the objective of making the best use of current technology'.

For some time, I had found VLSI associative memories intriguing and had deployed them in various cache memories and the like. But standard associative memories are 'brittle'; given exactly the right input, they produce exactly the expected output, but any error in the input results in no sensible output whatsoever. I figured that the brain is rather more flexible than this and can produce sensible outputs even when the input is a bit noisy, so the research question was to ask: could we reproduce this flexibility in an electronic system?

1.2.3 Reinventing Neural Networks

As we looked into the design of associative memories that were tolerant of input noise, every way we looked at this just seemed to be reinventing neural networks. I guess this isn't too surprising since a neuron can be viewed as an approximate pattern-matching device, at least in some modes of operation. After a year or two resisting this conclusion, we threw in the towel! If all roads to soft association lead to neurons and the brain, maybe we should turn to the brain for inspiration to guide our quest?

This line of thinking drew us inexorably towards neural networks as the direction we should explore to seek answers to our questions about how the brain does some things so much better than our machines, however fast they might be.

We (the Advanced Processor Technologies group) were a bunch of computer architects and engineers with a solid research background in unconventional (especially asynchronous) computing, taking novel designs all the way down to very demanding silicon implementations. What could we bring to the neural network party?

1.3 The Architecture Comes Together

So, hard logic had drawn us into the neural network game. This is not, of course, virgin research territory; many had looked into VLSI implementations of circuits based upon our (limited) knowledge of how the brain works. The first steps were to look into what had gone before, what was known about the functions of neurons and synapses, and what were the main problems that had arisen in previous work. Then, the goal was to synthesise something from our basic research strengths that stood a reasonable chance of yielding a substantive contribution to the field, with sufficient differentiation from others' work.

1.3.1 The State of the Neuromorphic Art

A lot of previous work in what was known as neuromorphic computing was based upon the use of analogue electronic circuits to map neural and synapse equations directly into the circuit function. This was then combined with a digital communications approach to convey neural 'spikes' (action potentials) between neurons. Biological neurons communicate principally (though not exclusively) through the transmission of action potentials which, because of the way they are electrochemically regenerated as they propagate along the neuron's axon (the biological wire that conveys the output of one neuron to the inputs of the next neuron), carry no information in the size or shape of the spike. Thus, the spike can be viewed as a pure asynchronous event, ideal for digital propagation – digital circuits have a similar ability to regenerate pure signals as they propagate.

The communication of spikes in electronic systems can be problematic, even when they are represented as digital signals, if the electronic system follows the example of the biological system in allocating an individual wire to each spike source. This had been resolved using Address Event Representation (AER) [152], where each spike source (neuron) is given a unique code or 'Address', and this address is then sent down a shared bus. Such an approach exploits the fact that, although electronic wires (which are restricted by manufacturing processes to two dimensions) are smaller in number than biological wires (which have the luxury of three dimensions to route through), they are also much faster. The only compromise inherent in this multiplexed approach to spike communication is that spikes that happen at exactly the same time in biology must be serialised in electronics, but the timing errors introduced by this compromise are too tiny to matter on biological timescales.

AER is a fine solution to spike communication up to the point where the shared bus begins to saturate, but it isn't scalable beyond that point. Biological systems can have immense numbers of neurons – the human brain has approaching one hundred billion – so something more than AER is required if we are to approach such scales of networks.

1.3.2 What Could We Bring to Neuromorphics?

The Advanced Processor Technologies group in the Department of Computer Science at the University of Manchester, had, as its core skill set, proven abilities in taking novel digital architectures and processing systems (generally based upon the ARM instruction set architecture) down to silicon implementation. We had limited experience in analogue circuit design, but we had done a lot of work on asynchronous digital circuits, where time is analogue (rather than discretised by an external clock signal) and signal levels are discrete (binary 0 or 1). As the brain operates without a central regulatory clock, would our experience in asynchronous design give us new insights into potential brain mechanisms?

Our digital design background naturally led us to think about digital implementations of the neuron and synapse equations. The major data structures required to model a neural network are those required to store synaptic weights that capture the strengths of connections between neurons. The obvious technology to store synaptic weights is some form of digital memory – if there's one thing that the semiconductor industry knows how to make it is memory! But memory is inefficient in tiny units, which led us to think about populations of neurons rather than individual neurons. If you think about the connections from one population of, say 256 (computer scientists do *so* like powers of 2!) neurons to another population of the same size, then those connections can conveniently be seen as a 256 × 256 matrix of, say, 8-bit weights, which can be implemented efficiently in 256 × 2 k digital memory array. A population of neurons with inputs from multiple other populations could use one such matrix memory for each population input.

So far so good. But how could we make the interconnect on a chip containing many such units flexible so that it could model an arbitrary range of network topologies? AER suggested a starting point but, as noted above, bus-based AER has limited scalability.

1.3.3 Multicast Packet-switched AER

The first key insight into the fundamental innovation in SpiNNaker was to transform AER from a broadcast bus-based communication system into a packet-switched fabric. In a packet-switched system, the AER information can be sent only where it is needed. As the system scales up, there is more communication resource available to handle the increased communication utilisation of the larger system. The scaling isn't linear, but on the assumption that the scaling laws for biological brains are similar to those for logic circuits as defined by Rent's rule [135], and subsequent evidence has emerged to support this supposition [10], then the traffic per unit of communication resource will grow much more slowly than the linear growth in a broadcast system.

That's all very well, but how are AER packets to be routed in such a system? The assumption at the time (again, subsequently supported by evidence [258]) was that a form of multicast routeing would be required to model the very high degree of connectivity found in biological brains, where neurons can have tens (in some cases hundreds) of thousands of inputs, and their outputs must therefore connect to similar numbers of other neurons. So the AER packet could not possibly carry information about all of its destinations. Each packet router in the system would have to know where to send each AER packet.

In principle, this sort of routeing could be achieved using table look-up, but the table would have to be infeasibly large. For example, a system designed to support up to 4 billion neurons would require a 32-bit AER field, and the table would have to have 4 billion entries. Even in today's remarkable silicon technology, this is far too large!

The solution is associative memory (naturally!). If each chip in a large system has its own router, that router will only see a small subset of all of the AER codes in the system, so an associative look-up table can be configured only with those codes that will pass its way. Linked to the associative table, a similar sized Random-Access Memory (RAM) can hold a vector of destinations for each entry in the associative table.

1.3.4 Optimise, Optimise...

If the associative table must have an entry for each neural AER code that comes its way, it might still need to be infeasibly large. But if we consider a population, not a neuron, as the basic unit, then all the neurons in a population will be routed to the same target populations, so cannot they be routed together using just one table entry? Indeed they can, if the associative table is implemented using Ternary

Content-Addressable Memory (TCAM), where each bit can match on 0, or on 1, or be ‘don’t care’.

What happens to AER packets that don’t match any TCAM entry? They can simply pass through a chip as if it weren’t there, so long straight paths through the fabric will not eat up precious table entries.

Now (in 2002) we had all the essential properties of SpiNNaker’s unique communications infrastructure: an AER-based multicast packet-switching system based upon a router on each chip that uses TCAM lookup for efficient population-based routeing and default routeing for long, straight connections.

1.3.5 Flexibility to Cope with Uncertainty

So far, we were still thinking in terms of implementing the neuron and synapse equations in digital hardware. But what is the right neuron model – there were several competing models at the time. And what is the right synapse model? Where does learning come in? Is there a consensus on learning rules? (No!).

I served as the Head of the Department of Computer Science, University of Manchester, from 2001 to 2004, and in academic year 2004/2005, I took a post-headship rehabilitation sabbatical. As part of the sabbatical year, I spent a couple of months at Sun Labs in Menlo Park, California, principally to work with their asynchronous design group led by Ivan Sutherland. But while there I was also asked to look at an emerging design for a High-Performance Computer (HPC), and in my own time, I also continued to think about SpiNNaker. These intermingled thought patterns led me to the firm conclusion that the way to accommodate the uncertainty in neuron and synapse models was to relegate these to the most flexible technology known to humankind – software. It was at that point that the digital hardware implementation gave way to programmable cores – ARM of course. These could be very small and efficient, as the neural modelling problem is embarrassingly parallel, so we could use lots of them.

The other thing I learnt at Sun is that HPC designers are paranoid about the possibility of deadlock caused by cyclic dependencies in the HPC communications fabric, so they avoid them like the plague. The techniques they use to avoid cycles incur considerable complexity and cost. The proposed SpiNNaker fabric was rife with potential cycles, but we couldn’t afford the HPC solutions in terms either of manpower or of silicon area! Neither could we afford deadlock. But each packet conveys only one spike, so do we really have to guarantee its delivery? The solution, which is simple and cheap, is *in extremis* to drop packets that cannot progress, thereby averting deadlock.[1]

[1] We subsequently added a mechanism to re-insert the dropped packets, re-establishing reliable delivery under most circumstances.

1.3.6 Big Memories

At this stage, we have a workable communications strategy and flexibility in neural and synaptic algorithms (through the use of software), but the memory requirements for the ARM cores were looking much too high. Ideally, each ARM should have its own memory for synaptic weights, but if it is to model a population of 1,000 neurons with 1,000 inputs, it will need a megabyte or so of memory. In Static Random-Access Memory (SRAM), this would be most of a chip. There was an alternative technology – Synchronous Dynamic Random-Access Memory (SDRAM) – that is much denser than SRAM, but integrating it onto the same chip as the ARM processors was problematic as DRAM uses a different process technology from logic/processors. A compromise – embedded DRAM – was on the scene, but the compromise included much reduced bit density.

The pragmatic solution was to use a separate industry standard SDRAM part which, although very cost-effective, also involved compromise. First, all of the ARM cores on the chip would have to share this memory, in terms both of space and bandwidth. Second, the data width available from a separate memory chip is much narrower than that which could be implemented on the same chip. All of this would lead to significantly higher power to work the memory interface at high speed, instead of having a very wide, slow and efficient on-chip interface. But engineering is the art of the possible, and the numbers worked, so the standard Synchronous Dynamic Random-Access Memory (SDRAM) became the memory of choice.

1.3.7 Ready to Go

This, then, was the thinking that went into defining the architecture of the SpiNNaker system – a processing chip with as many small ARM cores as would fit, each with local code and data memory, and a shared SDRAM chip to hold the large synaptic data structures. Each processor subsystem would have a spike packet transmitter and receiver and a Direct Memory Access (DMA) engine to transfer synaptic data between the SDRAM and the local data RAM, thereby hiding the variable SDRAM latency. Each chip would have a multicast AER packet router using TCAM associative lookup and links to convey packets to and from neighbouring chips.

All that was left to do was find funding to get the chip designed and built, then build a machine and, of course, write some software!

The following section reproduces the content of a note written in May 2005 that outlines the key architectural concepts that were the starting point for the SpiNNaker development. Some details changed in the course of that development, so this should be read as a historical note, not as an authoritative definition of the final architecture! Although some of the details would change during the implementation phase that followed, the key concepts are already in place in this note.

1.4 A Scalable Hardware Architecture for Neural Simulation

Steve Furber – 8 May 2005

1.4.1 Introduction

Over the last couple of years, I have been struggling with several aspects of the proposed neural hardware system. Issues that have come to the fore are the importance of modelling axonal delays, the importance of the sparse connectivity of biological neurons, the cost issues relating to the use of very large on-chip memories, and the need to keep as many decisions open for as long as possible. I have now found a way to resolve all of these issues at once through a radical change in the architecture proposal: push the memory off chip into a standard SDRAM and implement the on-chip neural functions through parallel programmable processors of a fairly conventional nature.

This approach yields a highly programmable system of much greater power than that previously proposed and a safer (more familiar) development path. It also points directly towards a development route that can be used to prove the proposed plan using technology already to hand.

1.4.2 Intellectual Property

The aspect of the overall design that seems to survive all of my attempts to find problems is the neural event routeing mechanism. This system is very simple, yet different from earlier inter-processor networks in that it supports one-to-many communication, whereas most inter-processor networks support one-to-one and/or broadcast communication. We could consider filing a patent on this.[2]

1.4.3 Market Opportunity

The more folk I talk to in the neuroinformatics area, the more I get the impression that this is a technology whose time may be about to come.

[2] We did!

In the first instance, I see this as a system that is well suited to supporting research into complex neuro-dynamics, and I think this will be the primary market until/unless there is a breakthrough in our understanding. This system is well positioned both to expedite that breakthrough and to exploit the consequences of it.

It is also possible that there might be a market for this system as a general-purpose low-cost high-performance computer system. It has very high integer performance and could be well suited to code-cracking, database search and similar applications that do not need floating-point hardware. However, this will require further investigation.

Potential products include neural simulation software, chips, boards and full-blown systems. We could also sell time on systems.

1.4.4 System Organisation

As before, the system comprises a regular tessellation covering a square 2D area with neural processing nodes (see Figure 1.3), each of which now comprises a special-purpose System-on-Chip (SoC) and a single SDRAM chip. Each node has eight bidirectional communication channels that will probably be used to connect to the eight neighbouring nodes, including diagonal connections.

Other configurations, such as a 4D hypercube, are possible, but the 2D arrangement with diagonals is very easy to put onto a Printed Circuit Board (PCB) and works pretty well.

The billion neuron system will require of the order of 10,000 nodes and will therefore be built in a rack system on a large number of PCBs (e.g. 40 PCBs each containing a 16 × 16 array of nodes).

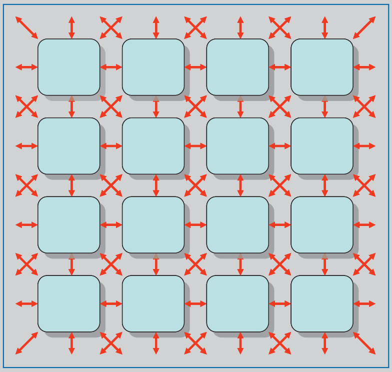


Figure 1.3 System-level organisation of nodes.

1.4.5 Node Organisation

The internal organisation of a node is illustrated in Figure 1.4.

This figure is, in fact, unchanged from the previous version. What it hides, however, is the fact that the implementation of the fascicle[3] processor is now completely different. This difference is highlighted in Figures 1.5 and 1.6.

Instead of each fascicle processor incorporating its own on-chip memory and hardware to implement the neural processing function, we now have off-chip memory in the form of a standard SDRAM that is shared between the fascicle processors.

This will allow the memory at each node to be increased by an order of magnitude (at lower cost), but presents an obvious bottleneck in the interface between the chips. Modern SDRAMs have very high bandwidth interfaces, but relatively long latencies. For this to work we will have to use the SDRAM very carefully, being sure to arrange data structures that are accessed together

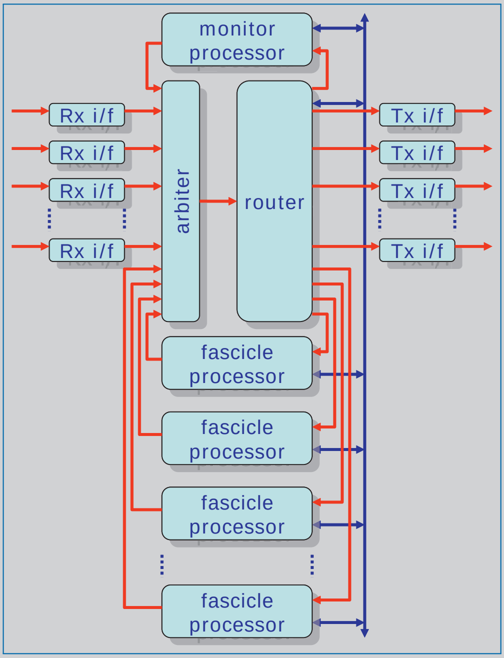


Figure 1.4 Node organisation.

[3] At this time, I had seen 'fascicle' used to describe a bundle of neuron fibres and thought it was widely used this way. I was wrong! We now use 'population' to describe a bunch of neurons with common inputs and outputs.

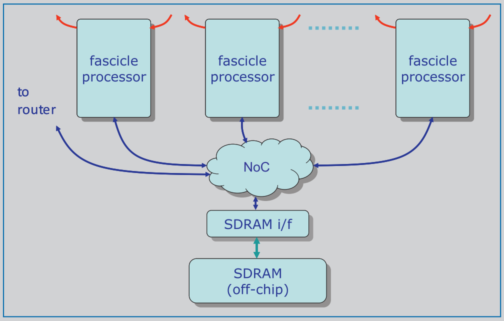


Figure 1.5 Detail of fascicle processor memory interface.

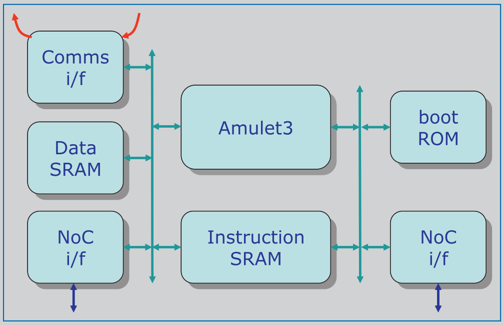


Figure 1.6 Detail of fascicle processor organisation.

in contiguous memory locations to make best use of streaming data performance, and so on. I believe that this can be made to work, but it is a critical design issue. In particular, it will be important to ensure that all of the instruction fetch activity of the processor is satisfied locally from the instruction RAM. I think this will be easy as the codes will be very small, but this can be checked by developing them in ARM assembly code. A few kBytes should be enough.

The fascicle processors share access to the SDRAM via a Network-on-Chip (NoC) along the lines of CHAIN, a very flexible interconnect technology proven on the smartcard chip (and currently being exploited by Silistix).

The internals of the fascicle processor node are illustrated in Figure 1.6.

Here we see the biggest change. In place of hardware neural processing logic, we have a conventional processor, for example, Amulet3 [267]. It is configured as a conventional system on chip with its separate instruction and data buses connected to a minimal set of local memories and communications devices. Each FP could have its own boot Read-Only Memory (ROM) (as illustrated), or there could be a single boot ROM shared via the

NoC – this needs to be investigated. The instruction SRAM is dual-ported solely to enable its contents to be modified by the processor itself.

1.4.6 System Architecture Issues

OK, so why does all (any?) of this make sense?

We have turned the neural model into software, making it fully reconfigurable, and providing scope for much more flexible axonal delay modelling, data structures that efficiently store sparse connectivity information, sophisticated Hebbian synaptic update rules and so on.

The design of the SoC is now a much more familiar job. We need to design the pretty simple inter-chip communication interfaces and router, an individual fascicle processor (which is then replicated many times on the chip), the NoC, and the SDRAM interface (which we may be able to buy-in – it's a conventional horribly synchronous block). This is still a big job, but it's a whole lot easier than what was proposed before.

The advantage of the SoC structure is that it is comprised principally of several copies of the fascicle processor. Even the monitor processor shown in Figure 2 can simply be another copy of this subsystem. This gives us a significant manufacturing cost advantage – if a fault causes one of these subsystems to fail, this does not greatly impair the usefulness of the chip; the rest of the system is likely to work fine. We just lose a bit of throughput.

The full billion neuron system is an interesting beast. It will require 10,000 SoC chips and the same number of SDRAMs. It will incorporate in the order of 200,000 processors between them delivering 64 Tera Instructions Per Second (TIPS). It will consume tens of kW. It will probably cost in the region of £0.5 million.

A single-board system comprising 256 nodes would simulate 25 million neurons and cost around £12,000.

Each node has about the performance of a Personal Computer (PC) in this application and is less than 10% of the price. In addition, PCs do not have the appropriate communication structures to enable a computing cluster to scale as well as the proposed design, so the price advantage at the high end is significantly greater.

1.4.7 Development Plan

This set of ideas is rather new so it will take some time to check that it is really feasible. In the meantime, there are several things we can do to check it out.

It is feasible to use the Excalibur parts we obtained from Altera to prove the ideas. These chips include a 200 MHz ARM9 with caches and an SDRAM interface, and an area of programmable logic. We can prototype the neural algorithms on the ARM9 and prototype the router and inter-chip communications in the programmable logic. We have 2 development systems to get started and 10 chips that could be used to build a 100,000 neuron engine. Such a system would be an asset if we wished to attract venture capital funding to support the SoC development and/or production.

Alternatively, I could put in a large EPSRC proposal to support the SoC design.

Timescales (rough estimates):

- 2 persons for 1 year to develop algorithms and Excalibur prototype (~£150 k).
- 4 persons/3 years to develop SoC design (~£900 k).
- Some significant amount of time for configuration software development.

This will yield prototype silicon. Moving this into production will incur a large mask charge ($1.5 million) and, at this stage, this will require a partnership and/or investment.

1.5 Summary

The above May 2005 note, reproduced in Section 1.4, outlines all of the key concepts at the start of the development of the detailed design of the SpiNNaker chip. Funding was successfully sought from EPSRC, and the design work started in earnest in October 2006. Many of the estimates in the note turned out to be horribly optimistic – for example, the chip design took more like 5 years and 40 person-years rather than the 3 years and 4 person-years (4 years and 6 person-years including the prototype, that was never built) in the note, but the choice of a 130-nm Complementary Metal Oxide Semiconductor (CMOS) technology kept the mask cost to $250 k rather than $1.5 M, so swings and roundabouts!

DOI: 10.1561/9781680836530.ch2

Chapter 2

The SpiNNaker Chip

By Jim Garside and Luis A. Plana

Architecture should speak of its time and place, but yearn for timelessness.

— Frank Gehry

The central component in the SpiNNaker system is the SpiNNaker chip [186], and the central focus of the SpiNNaker chip is *scalability*. The key concepts were described in the previous chapter, but now these concepts must be realised in practice. This realisation, which took 40 person-years of design effort and 5 years of elapsed time, is the subject of this chapter.

2.1 Introduction

Biological neurons are fairly slow at processing. The processes they perform are quite complex and the appropriate abstraction – to separate the computing from the process of simply living – is unclear, although the models are becoming more sophisticated annually. There are also a *lot* of neurons in a mammalian brain and, despite dense connectivity, most are independent from each other.

Electronic computing devices are very much faster than biology at computing simple functions. This means that one electronic device can, in principle, model numerous biological neurons and still provide real-time performance. There are many possible levels at which a model can be built, ranging from direct electronic models of the neurons (which can process many times faster than biology) [114] to massive computers that trawl through enormous data sets at great speed [199]; each approach has its merits and demerits.

SpiNNaker [65] was designed to function somewhere in the middle of this spectrum. To provide the flexibility to *experiment* with neuron models, it was determined that these should be implemented in software. Running software carries a significant overhead in both performance and power consumption: the former can be addressed by using a large array of processors, since the problem is amenable to a massively parallel-processing solution; the latter concern was tackled by employing power-efficient rather than fast microprocessors.

2.2 Architecture

2.2.1 An Overview

Imagine a large array of microprocessors where each processor simulates the biological computing of a number of neurons. In imagination, the array is almost infinitely scalable, since the neurons themselves are largely independent. There is then a choice as to how many neurons are mapped onto each processor, which is governed by speed – both of the processors themselves and the desired speed of simulation – and the memory capacity of each processor.

Outside the world of imagination, there are other pragmatic limits. Building a customised microprocessor, specialised for neuron modelling, is impractically expensive, not (just) from the hardware development view but from the software support: an established architecture is much to be preferred. Then, there is the consideration of powering and cooling a machine of any size. Finally, if a custom logic is to be made, the design and verification effort must not be impractically high.

To provide significant (and convenient) computing power without excessive electrical power dissipation suggests a 32-bit architecture. A 32-bit integer can provide 2^{32} or about four billion unique codes, which is (very) approximately a match for the number of neurons in a mammalian brain. (A human has about 86 billion neurons; a domestic cat has around three-quarters of a billion [19, 90].) As a *back of the envelope* initial figure, 'one billion neurons' seemed a credible target. This could be spread over a million processors – each simulating 1,000 neurons – with the processors grouped into chips, each chip being a multicore Application-Specific Integrated Circuit (ASIC).

The chosen processor was an ARM968 [6]. This ARM9 device was already *mature* at the time of selection but still gave good power/performance efficiency and, crucially, was kindly licensed, on a non-commercial basis, by ARM Ltd. For manufacture, a 130 nm process was selected: again not state-of-the-art even at the time of design but cost-effective and without too many new process issues for the (necessarily) limited design team. With this process and this processor, a target operating clock frequency of 200 MHz seemed reasonable and static RAM macros that supported this target were available. Calculation suggested that this could support the target number of neurons in real time, with some flexibility to cope with a varying load. Energy efficiency is important not so much on an individual processor basis but when multiplied by a million processors in the system or, indeed, twenty or so in the same package; the ARM968 is a power-efficient microprocessor when executing and is able to 'sleep' – consuming almost no dynamic power – when there is nothing to do, which may be expected frequently in a real-time system.

The amount of RAM needed to balance this model was also reckoned. In practice, for the intended application, the RAM was infeasibly large; however, much of this is relatively infrequently used, so the model was subdivided in a memory hierarchy, with a fast SRAM and a much larger but slower SDRAM component. A local data space of 64 KByte plus 32 KByte of code space (small, since the processors are running dedicated, embedded application code) was allocated. This needs to be backed up by tables up to *a few megabytes* in size. Available (low power) technology meant a single cost-effective die supplied 128 MByte but the relatively low demands expected meant that one die could reasonably be shared amongst several processors.

With area estimates for the processor subsystems – including their SRAM – and a feasible ASIC die size – it appeared that *about* 20 processors on each ASIC, together with a single, shared SDRAM, would provide an appropriately balanced system. This implied that 50,000 ASICs would be needed for a 1,000,000 processor machine – a number which would (attractively) fit in a 16-bit binary index.

Neurons alone do not compute; there needs to be interconnection and, indeed, there is overwhelming evidence that it is the patterns and strengths of *connections* which *programme* biological computers [115]. The problem for the system architect is that, in biology, the output from any one neuron may be routed to a unique set of hundreds, thousands and even tens of thousands of destination neurons (Figure 2.1). This far exceeds typical computer communications uses, other than with a broadcast mechanism; here, with a million possible sources, broadcast is not practical, either from the communications bandwidth needed or the power requirement for inter-chip communications.

It is therefore the specialist communications network, designed to support the specific spiking neural network applications, that differentiates SpiNNaker from most other multiprocessor systems.

SpiNNaker communicates with *short* packets. In neural operation, each packet represents a particular neuron firing. A packet is identified using AER [152]; it is tagged only with its originator's identifier. (With 1 billion neurons, this requires at

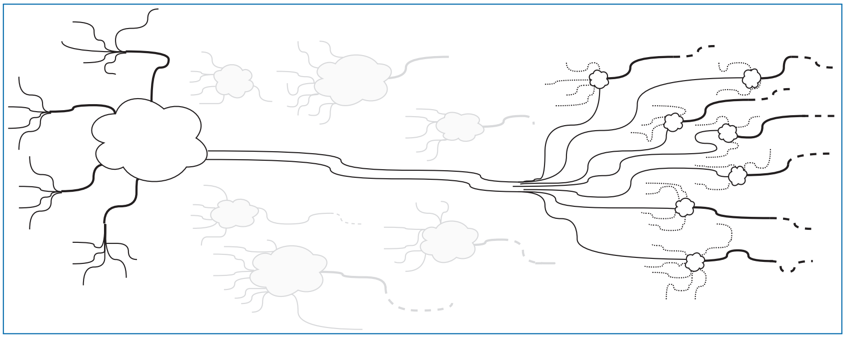


Figure 2.1. Neurons interconnecting.

least 30 bits; a 32-bit field is allocated for convenience.) Packets are then *multicast* to their destinations with most of the routeing and duplication being done in (and by) the network itself.

The first important point in the design is that the *aggregate* bandwidth of the running system – where packets are duplicated *in flight* but only as needed to reach all their destinations – is not infeasibly high. Just like the processor – neuron relationship, a single network link can carry many, multiplexed *spike* links as the electronic connections are much faster than the biological axons. Indeed, practically, the time to deliver a spike is typically negligible compared to biological transmission. Thus, the actual network topology is not particularly important although, since neural systems themselves (and their traffic) are fairly homogeneous, some form of mesh is suitable – and amenable to the construction of scalable systems.

The chosen topology for the SpiNNaker network is a two-dimensional mesh. The mesh is triangular (Figure 2.2) rather than Cartesian, with each ASIC connected to six neighbours; this provides more potential bandwidth over the given links and was also intended as a provision for automatically routeing around faulty connections. (In practice, it has been observed that this latter feature was over-cautious and is little used.) The edges of the mesh can be closed to form a torus that reduces the longest paths; the maximum expected system – 2^{16} chips or a 256 × 256 grid – would therefore have a longest path of 128 *hops* although most would be much shorter.

Although there are other packet delivery mechanisms, the novelty and specialisation in SpiNNaker is in handling *multicast* packets. These are optimised to model biological neuron interconnection, where each neuron has a single output that feeds its own set of targets. Biological destinations are not entirely random; there is some structure and neurons tend to be clustered within *populations* with an output feeding some subset of the neurons in several populations. This structure can be abstracted as a *tree* (Figure 2.3).

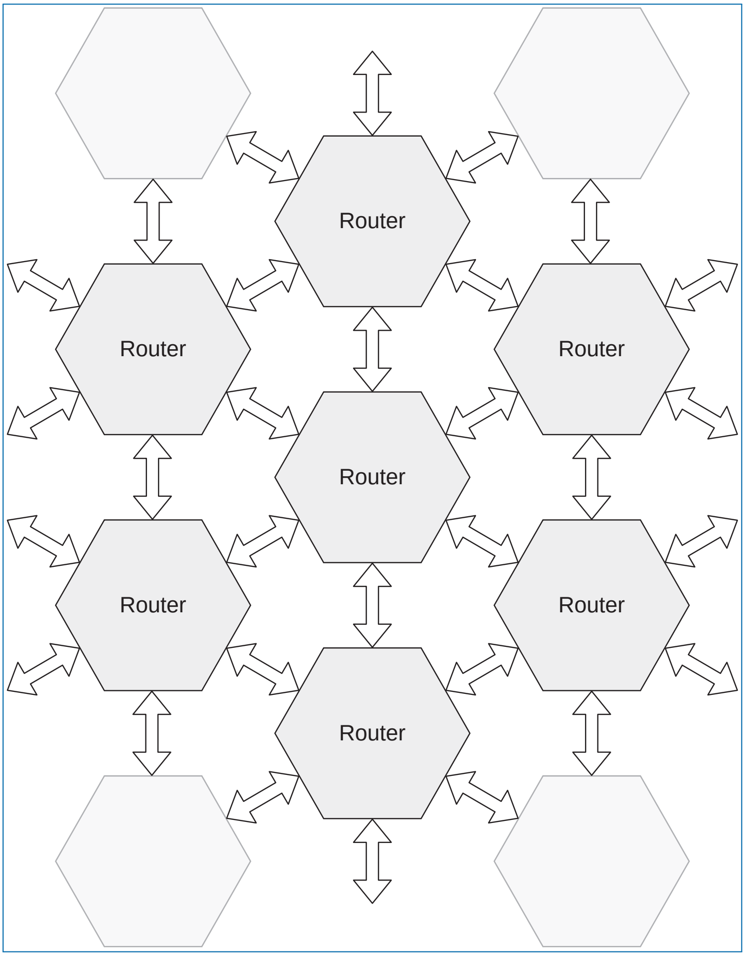


Figure 2.2. Network mesh.

For simulation, it is logical to map neurons within a population to the same processor(s). This means that a single packet delivered to a processor can be multicast to the neurons – the last branching of the tree – by software. The populations themselves need to be distributed across the mesh network. In this manner, it is likely that multicast packets can share part of their journey, effectively extending the tree structure to multiple (series) branches (Figure 2.4). This also reduces the network traffic as a packet is often not cloned until some way towards its destination.

The routeing from chip to chip is managed by a custom router on each ASIC. Logically speaking, each router checks the (neuron) source ID – the only information in the packet – and looks up a *set* of outputs, potentially including both chip-to-chip links and processor systems on that chip itself. The packet is then duplicated to all the specified outputs.

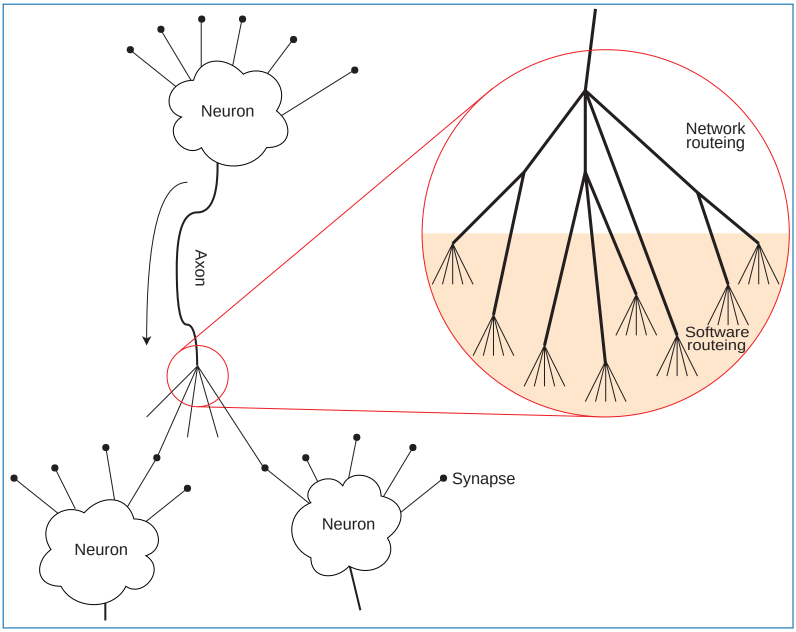


Figure 2.3. Neuron tree.

With a 32-bit neuron AER, each router is potentially holding 4 billion words of routeing look-up table: this is impractical. However, the logical table can be compressed considerably in practice:

- Not all IDs are expected at a given node.
- A high proportion of connections – particularly over long distances – are simply routed *straight through*.
- Many entries will be the same, as a result of the population-to-population connectivity, rather than a random structure.

These properties are exploited to shrink the routeing tables to a manageable size. This makes the table sparse, so rather than a simple array it is stored as an *associative* structure using Content-Addressable Memory (CAM) to identify IDs of interest. If an ID is *not* recognised, a topological assumption is made about the interconnection mesh and the packet is simply forwarded to the *opposite* link from which it arrived: this is referred to as *default* routeing (Figure 2.5). Default routeing reduces the number of table entries to those corresponding to packets which are both expected and need some action: changing direction in the mesh, being duplicated or arriving at their destination – or any combination of these.

Lastly, providing the neurons in a given population are identified *sensibly* – i.e., with similar IDs – they can usually be routed with a single table entry. This is

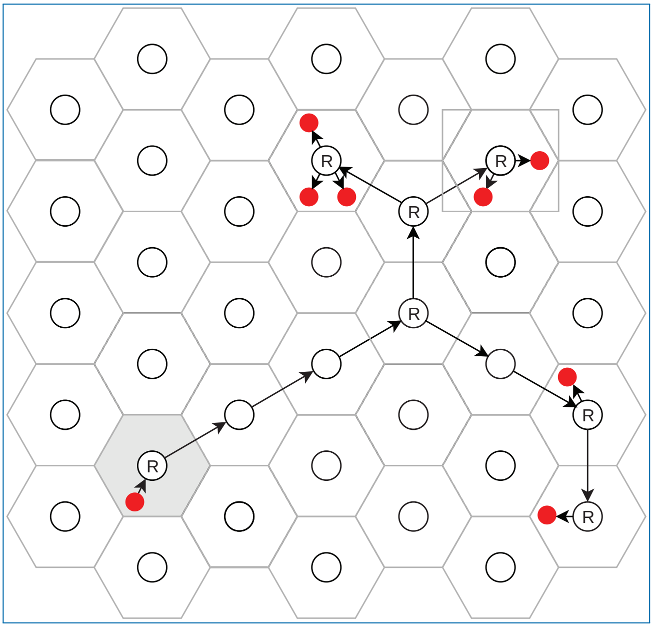


Figure 2.4. A single neuron tree mapped onto a SpiNNaker chip network. The source neuron is on the shaded chip. 'R' indicates a router table entry; other involved routers use *default* routeing. Solid dots are processors and spikes are typically duplicated to many neurons in each by software.

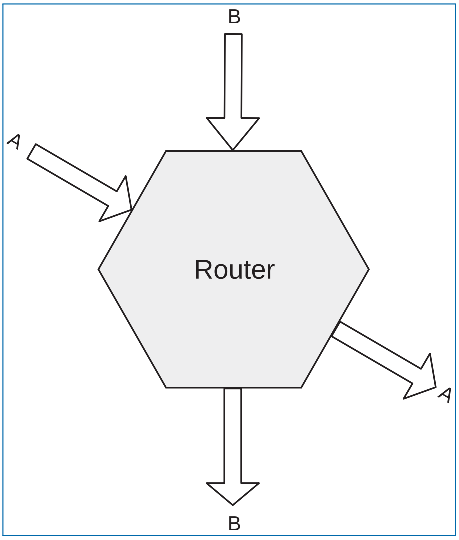


Figure 2.5. Examples of default routeing.

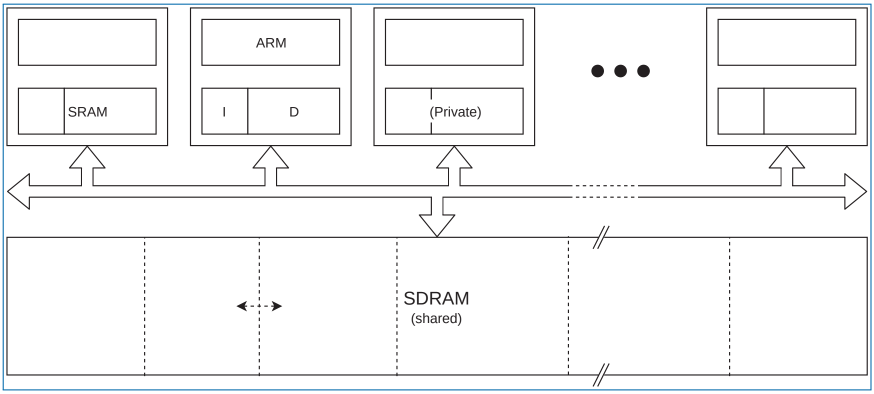


Figure 2.6. Memory *hierarchy* on a single processor.

because the CAM contains a binary mask that specifies which bits in each key are significant to that router. For example, if a population contains around 2,000 neurons, it can have a 21-bit ID with the remaining 11 bits determining the particular neuron. One routeing table entry can provide for all 2,000 neurons. For implementation, the number of table entries is arbitrary: 1,024 was chosen for SpiNNaker.

The final stage of neuron packet routeing takes place after delivery to a processor subsystem. Here a *spike* is multicast to a subset of the local neurons; however there is now more information needed. Each connection has some associated information:

- the strength or *weight* of the connection
- the unique delay of that connection, simulating the biological connection delays.

The details of these variables are not important here. What does matter is that there is one entry per *synapse*. Even with a very rough calculation – say 1,000 neurons each with 1,000 synapses – it becomes clear that several megabytes of storage are required for each processor subsystem. This is the data that reside in the (shared) SDRAM and is fetched on demand.

Each processor has its fast, private memory and shared access to the SDRAM (Figure 2.6). Although it can be used for communications, the main intended purpose of the SDRAM is to act as a backing store for the large, relatively infrequently accessed data tables. For this purpose, the SDRAM space is partitioned in software with each processor allocated space according to its needs. For many applications, data are simply copied in as needed although synaptic weights could be modified and written back if the network is adaptive.

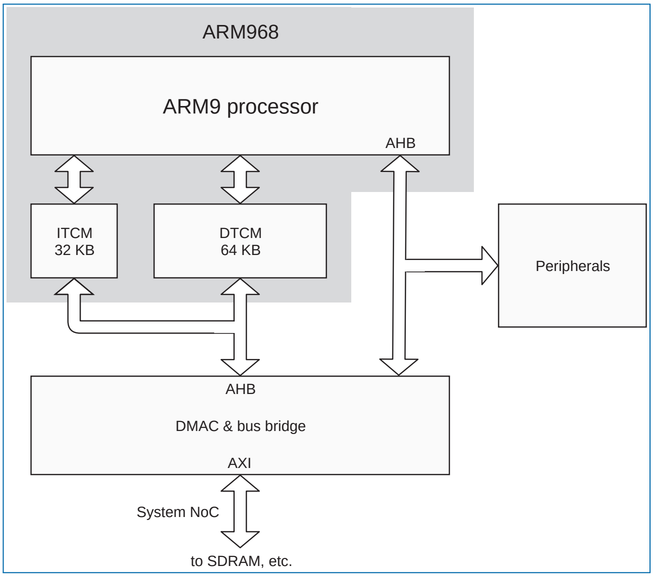


Figure 2.7. Processor subsystem block diagram.

The act of moving data around the memory map is simple but tedious and inefficient for software. Each processor subsystem therefore contains a memory-to-memory DMA Controller (DMAC) that can download these structures in the background. The unit is also capable of uploading data if the synaptic weights change, which will occur if the neural network is *learning*. The impact of transfers on the processor is minimal since the local SRAM is bank-interleaved, always assuming the processor has other work to do.

The impact of DMA transfers on the *processing* should also be small as the fetching of data is a background task. To decouple the process further, the DMAC has a command buffer, allowing a request to be queued while its predecessor is in progress; DMA transfers can therefore run continuously (if necessary) with considerable leeway in servicing the *completion* interrupts.

Other than the ARM968, its RAM and the DMAC, there is very little else within a subsystem. The only peripherals are timers, a communications interface, which allows the processor to send and receive packets and an interrupt controller (Figure 2.7).

The ASIC was planned to contain *about* 20 such processor systems. All the processor subsystems used identical layout for development convenience, meaning the timing closure was only necessary once; on the chosen manufacturing process, it is permissible to rotate, as well as reflect, the *hardened* layout macrocells. When possible floor plans were examined and the feasible chip area was taken into consideration, it became apparent that 18 processor – memory combinations,

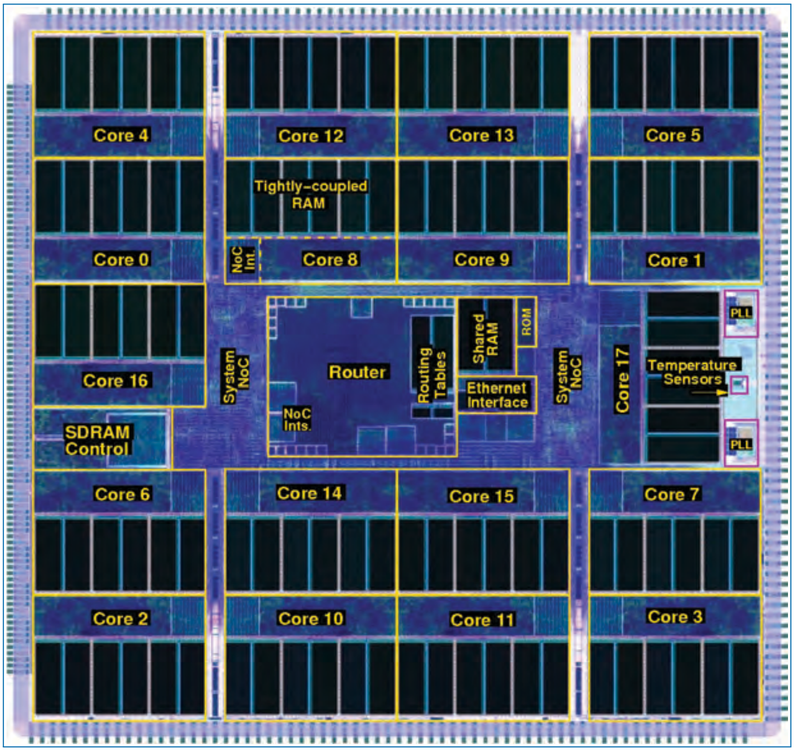


Figure 2.8. The SpiNNaker chip floor plan.

together with the router, fitted better. As the specific number was not critical, this was adopted (Figure 2.8). This can be post-rationalised into 16 neuron processors, a *monitor* processor to manage the chip as a computer component plus a spare, but the constraint was primarily physical. The processor count does have some impact on the router since, when multicasting packets, it is necessary to specify whether *each* of the 24 destinations – 6 chip-to-chip connections plus 18 local processors – is used; 24 bits is a reasonably convenient size to pack into the RAM tables, so this is a bonus.

There are also a few shared resources on each chip, to facilitate operation as a computer component. These provide features such as the clock generators; interrupt and watchdog reset control and communications; multiprocessor interlock support; a small, shared SRAM for inter-processor messaging; an Ethernet interface (as a host link) and, inevitably, some general purpose I/O bits for operations such as run-time configuration and status indication. A boot ROM containing some preliminary self-test and configuration software completes the set of shared resources. The details of some of these components are discussed in the following sections.

2.2.2 Processor Subsystem

The ARM968 is, by current or even design-contemporary comparison, a fairly low-performance 32-bit processor. However, it is also power-efficient, which was a primary design criterion. The intention was to keep the ASIC power below or around 1 W or about 50 mW per processor subsystem – individually quite small but still resulting in a 50 kW dissipation in the full-scale machine. The target speed was 200 MHz operation although, as SpiNNaker is intended as a real-time system, the processors can be halted much of the time. Leakage power is effectively negligible in this 130 nm process, so halting a processor drops its power dissipation to near zero.

The ARM968 is an integer-only processor. For neural calculations, the expectation was that some *noise* was expected – biological components are not 100% reliable, after all – so the overhead of a floating-point accelerator was regarded as decadent. It has no cache either, but does support Tightly-Coupled Memory (TCM) on both its instruction and data buses. The TCM is a static RAM with single-cycle access; the processor can perform parallel instruction and data accesses. It acts like a cache memory except it is under software control. Although the processor can address shared memories directly – a boot ROM, an on-chip (32 KByte) SRAM and the SDRAM – these are very slow in comparison so all applications code is kept in the 32 KByte Instruction Tightly-Coupled Memory (ITCM) and the data working set and stack in the Data Tightly-Coupled Memory (DTCM).

The TCM is *logically* dual-ported, so it can be written and read by the DMAC too. The ITCM comprises a single 8 Kword SRAM block, so DMA updates will interfere with instruction fetches, slowing down access somewhat; however, code updates while running applications are not usually anticipated. The DTCM comprises two word-interleaved SRAM blocks, so DMA updates – which are expected in the background – are minimally intrusive. The result is a close-to-zero-wait-state access for the TCM, despite it being composed from standard single-port SRAM macrocells (Figure 2.9).

In addition to its TCM buses, the ARM968 has a separate AMBA High-Performance Bus (AHB) that handles the remaining addresses. As a microcontroller processor – and in its era – it is clear that this processor was never intended to have a completely filled address space. This is evident in the way the address space is decoded as it is divided into 256 MByte regions which are alternately write-buffered and unbuffered. In some places, this is exploited in the SpiNNaker memory map (Table 2.1), so that (for example) the 128 MByte SDRAM is aliased into two places. In most cases, the write-buffer can be exploited to give better memory throughput; however, for barrier operations, where it is important to know that writes have completed, and most I/O operations, the unbuffered space should be used.

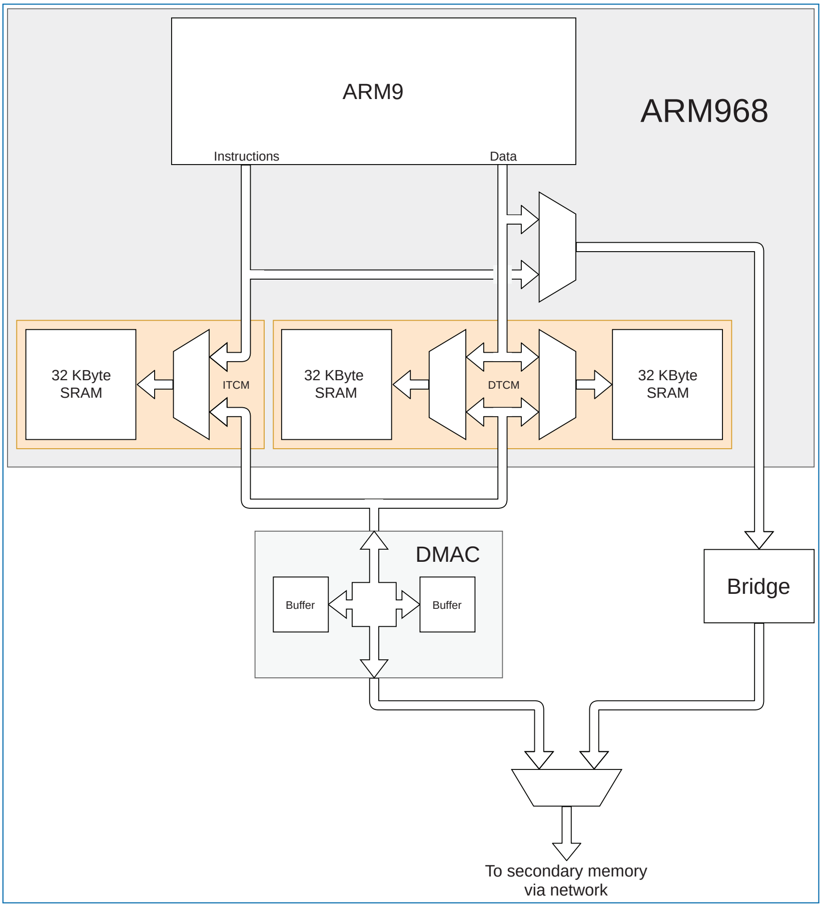


Figure 2.9. Memory structure.

The DMA Controller (DMAC) is programmable via the ARM968's AHB, in addition to being a bus master to the TCMs (Figure 2.7). The DMAC is capable of memory-to-memory transfers between the TCMs and the chip wide bus network – primarily the shared SDRAM. Transfers are in 32-bit words and can be of any length; they are subdivided into bursts that form small block transfers using an Advanced eXtensible Interface (AXI) protocol. Contiguous data bursts make more efficient use of both the AXI interconnection and the SDRAM itself than a set of individual transfers. The bursts are buffered by the DMAC, so both buses can be active simultaneously: for example, an SDRAM read can be incoming to a buffer, while its predecessor is being written to TCM. Transfer requests are buffered so that a new transfer may begin as soon as its predecessor completes.

Table 2.1. Memory map.

Start	End	Actual size	Access	Function
0000_0000	003F_FFFF	32 KByte	Local	ITCM (instruction memory)
0040_0000	007F_FFFF	64 KByte	Local	DTCM (data memory)
0080_0000	0FFF_FFFF	16 MByte	Local	ITCM/DTCM (alias)
1000_0000	10FF_FFFF	32 Byte	Local	Communications
1100_0000	1EFF_FFFF	4 KByte	Local	Counter/timer
1F00_0000	1FFF_FFFF	4 KByte	Local	Interrupt controller (alias)
2000_0000	20FF_FFFF	32 Byte	Local	Communications (alias)
2100_0000	2EFF_FFFF	4 KByte	Local	Counter/timer (alias)
2F00_0000	2FFF_FFFF	4 KByte	Local	Interrupt controller (alias)
3000_0000	3FFF_FFFF	512 Byte	Local	DMA controller
4000_0000	4FFF_FFFF	512 Byte	Local	DMA controller (alias)
5000_0000	0000_7FFF	–	None	Bus error
6000_0000	67FF_FFFF	128 MByte	Global	SDRAM (buffered)
6800_0000	6FFF_FFFF	128 MByte	Global	SDRAM (alias)
7000_0000	77FF_FFFF	128 MByte	Global	SDRAM (unbuffered)
7800_0000	7FFF_FFFF	128 MByte	Global	SDRAM (alias)
8000_0000	DFFF_FFFF	–	None	Bus error
E000_0000	E0FF_FFFF	4 KByte	Global	SDRAM controller (alias)
E100_0000	E1FF_FFFF	96 KByte	Global	Router (alias)
E200_0000	E2FF_FFFF	516 Byte	Global	System peripherals (alias)
E300_0000	E3FF_FFFF	4 KByte	Global	Watchdog (alias)
E400_0000	E4FF_FFFF	48 KByte	Global	Ethernet (alias)
E500_0000	E5FF_FFFF	32 KByte	Global	Shared SRAM (buffered)
E600_0000	E6FF_FFFF	32 KByte	Global	Shared ROM (alias)
E700_0000	EFFF_FFFF	–	None	Bus error
F000_0000	F0FF_FFFF	4 KByte	Global	SDRAM controller

(*Continued*)

Table 2.1. Memory map (continued).

Start	End	Actual size	Access	Function
F100_0000	F1FF_FFFF	96 KByte	Global	Router
F200_0000	F2FF_FFFF	516 Byte	Global	System peripherals
F300_0000	F3FF_FFFF	4 KByte	Global	Watchdog
F400_0000	F4FF_FFFF	48 KByte	Global	Ethernet
F500_0000	F5FF_FFFF	32 KByte	Global	Shared SRAM (unbuffered)
F600_0000	F6FF_FFFF	32 KByte	Global	Shared ROM
F700_0000	FEFF_FFFF	–	None	Bus error
FF00_0000	FFFF_FFFF	16 MByte	Global	Boot area
FFFF_0000	FFFF_0FFF	32 Byte	Global	Boot vectors
FFFF_F000	FFFF_FFFF	4 KByte	Local	Interrupt controller

It was also anticipated that in an expanded system, the soft error rate in the aggregate SDRAM would be non-negligible. The DMAC therefore includes a programmable Cyclic Redundancy Check (CRC) generator/checker that can append a CRC word when a transfer is written to SDRAM or verify a CRC when it is read.

Also contained *within* the DMAC, although not a DMA function, is a bus bridge that allows the ARM direct access to the SDRAM, although this form of access is not particularly efficient. A write buffering option is available to reduce the latency if desired.

The only peripheral of particular note is the communications controller. This provides bidirectional on-chip communication with the router. The input interconnection is *blocking*, so it is important to read arriving packets with low latency; the ARM's Fast Interrupt Request (FIQ) is typically used for this. Failure to read packets will cause the appropriate network buffers to fill and, ultimately, stall the on-chip router. Similarly, the outgoing link is blocking but the back-pressure may partially rely on software checking availability.

2.2.3 Router

The router is the key *specialised* unit in SpiNNaker. Each router has 24 network input and output pairs, one to each of the 18 processor subsystems and 6 to connect to neighbouring chips. Largely the links are identical, the only difference being that off-chip links (only) are notionally *paired*, so that there is a default output associated with each input which is used in some cases if no other routeing information is found.

All router packets are *short*. They comprise an 8-bit header field, a 32-bit data field and an *optional* 32-bit payload. Much of the network is (partially) serialised, so omitting the payload when not required reduces the demand on bandwidth and saves some energy.

There are four types of packet:

- Multicast (MC) packets are intended to support neural spike communications.
- Point-to-Point (P2P) packets are for chip-to-chip messages and are intended for machine management.
- Nearest Neighbour (NN) packets primarily support the machine boot and debugging functions.
- Fixed Route packets contain no key information and are always routed the same way: they can provide facilities such as carrying extra status data to a host.

Each of the packet types is separated and routed according to its particular rules. The simplest are P2P packets that provide *chip* interconnection. A fully expanded SpiNNaker system is designed to have 2^{16} chips, so a 16-bit field in a P2P packet determines the *destination chip*. This is used as an index into a RAM table that specifies which output link to use for that packet. Each entry in the table is 3 bits long, which permits the selection of any of the six chip-to-chip links plus an *internal* option, used for when the packet has reached its destination chip; the routeing of all possible packets is therefore fully specified in this table.

When the P2P packet reaches its destination chip, it has to be directed to a particular processor. All *internal* P2P packets are sent to a preselected processor subsystem, programmed into that router. The design intention is that this *monitor* subsystem will, at least primarily, manage the computer itself rather than run applications. It can forward messages to other systems if required in software, using the shared RAM on the chip.

MC packet routeing is rather more complicated. As previously mentioned, it is not feasible to store a complete routeing table for a billion neurons, so the neurons are grouped and only a subset of the groups need be recognised by any particular router. The first job is to recognise a packet (or not). This function is performed by a TCAM in which the packet key is compared with *all* the entries. Each table entry consists of a *key* and a *mask*. Within each entry, each bit is compared with the corresponding stored state, which can be:

Mask bit	Key bit	Function
0	0	Always match
0	1	Never match
1	0	Match if 0
1	1	Match if 1

Subsequently, all the bit *matches* are ANDed, and if the result is true, the entry is a 'hit'. These combinations allow each entry to match with particular patterns of '0's and '1's in the key, disregarding some other bits. For example, an entry with *key* = **0x5a5a5a00** and *mask* = **0xffffff00** will match the 256 packet keys in the range [**0x5a5a5a00**, **0x5a5a5aff**] as it ignores the 8 least-significant bits. Including a *never match* bit anywhere in the entry indicates that the entry is unused, as it will never produce a match.

The inclusion of *don't care* fields means that it is possible to match multiple different TCAM entries quite legitimately. This is an exploitable feature since the matches are prioritised and the highest priority match is isolated for the subsequent stage. Placing more specific entries in higher priority positions can simulate having more entries than are physically present. For example, an entry with *key* = **0x5a5a5a5a** and *mask* = **0xffffffff** will match the single packet key **0x5a5a5a5a**, which is part of the range matched by the entry listed in the previous paragraph. If the new entry is included in the table at a higher priority than the previous entry, it will make that entry only ever match the other 255 keys in the range. Matching a set of 255 packet keys would require a larger set of non-prioritised entries.

If a match has been made, the next step is to look up the output vector. This comprises a 24-bit word where each '1' bit indicates that the packet should be copied onto that link. This facilitates the multicast operation.

Fixed route packets are very simple to direct. Each router has a single, programmable register that says which output link(s) to use. They are really a special case of MC packets with a single, *always matched* key field and they require almost no additional hardware. They can be used for specific purposes, such as building network trees to funnel monitoring data back to host interfaces but can only provide one such structure in any single configuration.

Unlike the other communication packets, NN packets can be routed before the network tables are initialised; their routeing is determined by the chip hardware and the network topology. They are provided:

- for boot purposes
- for local systems communication
- as a debugging aid.

For the first two purposes, packets are routed:

Source	Destination
Any processor on this chip	One or all inter-chip links
inter-chip link	monitor processor on this chip

By convention, only the local monitor processor should originate such packets; just like the other packets, they carry a 32-bit data field with an optional 32-bit extra payload.

For debug purposes, a different type of NN packet is used. These are trapped by the router on the destination device, which becomes a *master* of the shared address space on that chip. This means that one chip can read and write some of the state of any neighbouring device. The convention adopted here was that only 32-bit words can be moved and the presence of a payload: in a request indicates a write request; in a response indicates a returned read value.

All the routeing units deliver packets to an output stage together with a *bit vector* indicating their output direction(s). All being well, copies are dispatched simultaneously on each of the indicated links. However there can be congestion which causes *back-pressure* on an output; in this circumstance the router output stalls and waits for the link(s) to clear. MC packets stall if *any* output is blocked rather than transmitting on the unblocked links first; this facilitates some error recovery, if necessary, later.

The network is *not* guaranteed deadlock free! In particular, the cloning of MC packets can generate a lot more traffic than is initially injected. It is also infeasible to implement an end-to-end flow control protocol on such packets. There is therefore a risk – indeed a significant probability! – that the network could deadlock, at least unless some other protection exists. This contingency is handled by using a time-out on blocked packets. If a packet has been stuck for a pre-programmed time, it is *dropped* and the next is output instead. Dropped packets are caught in software and can be re-injected later. Ensuring that (multicast) transmission is all-or-nothing means that only the packet needs to be saved, the packet routeing being re-derived on re-injection.

The time-out period is software programmable using a short (8-bit) *floating-point* value, allowing times from 0 (i.e. discard immediately if an output is blocked) to almost a million clock cycles; the ultimate value is *wait forever* for experimental purposes: packets are not lost but, under many circumstances, system deadlock means that functionality is!

The time-out introduces elastic buffering at a router and can alleviate any conceived deadlock problem. In a case of severe congestion, it does impose a potential performance penalty in that the time-out has to be slow enough to allow a processor to respond to an interrupt and read the packet before the next one is dropped, limiting the minimum time-out interval. In the original conception, it was believed that for neural spike packets, the vast majority of traffic in the intended applications could be dropped without re-injection because it is likely that biological systems would tolerate such *noise*. In hindsight, this was a questionable decision since computational neuroscientists can be more protective about their simulations!

Another level of defence against faults was provided by *emergency routeing*. This is a mechanism primarily to protect against a physical link failure. At time-out, instead of immediately dropping a packet, the router can attempt to route around the blockage. This assumes that the inter-chip mesh is wired in a particular (i.e., the originally intended, triangular) way and uses an alternate, two-hop path to the destination (Figure 2.10). Information is included in the header of any packets treated this way, so that the subsequent routers can allow for the diversion, including knowing the appropriate *default routeing* direction. The emergency routed packet is, in some sense, *superimposed* on the network, so that the case where it exploits a link *which was already committed to a valid route* is also correctly handled. Packets can also stall waiting for emergency routeing to be available; a second (independently programmable) time-out mechanism is applied which, if set to zero, can effectively disable emergency routeing.

In addition to its routeing links, the router has an AHB slave interface for programming and monitoring. This allows the tables to be set up. Although this is normally the responsibility of the local *monitor* ARM, it is also possible using NN packets and the router as its own bus master. It is possible, with care, to read and write the look-up tables while the system is operating since requests are arbitrated into the packet stream to reach the TCAM, RAMs, etc. and removed before the output stage when a response can be generated. This also facilitates testing of the RAMs. Testing the TCAM is somewhat more complicated since it is not directly readable.

Neither full custom design nor TCAM macros were feasible or available for the design. While the RAMs could use conventionally produced (and efficient) macrocells, the TCAM is composed of standard cells. This means that it significantly dominates the silicon area occupied by the router. This cost was alleviated using

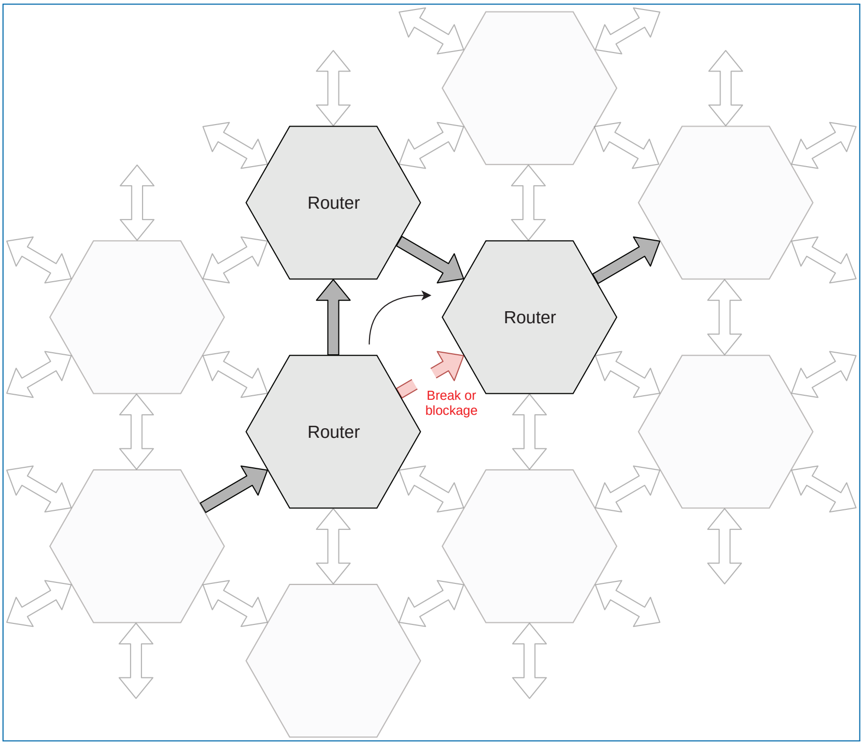


Figure 2.10. Emergency routeing.

latch, rather than D-type flip-flop, cells for storage, which roughly halves the area. To meet timing constraints, *writing* to these latches requires two clock cycles with a resulting *hiccup* in the pipeline flow; however, writing is rare, so this is not a serious issue. To further reduce cost, the multiplexer trees that would be needed to read back the contents were omitted. Some means of production test is still required though, and a scan chain through the latches is a difficult (and costly) alternative.

Instead, the TCAM is tested by association. A *key* pattern can be written to a test register location and the presence or absence of a match can be determined, together with the internal address of the first match. The test is conducted by one of the on-chip processors during the boot process.

In a fault-free environment, all packets arriving at a router will be intact, correct and intentionally present. However, the router does some straightforward checks to increase the robustness of the system. Firstly, an arriving packet has to have a legal size, as counted by the number of symbols ('flits') arriving, delimited by End-of-Packet (EoP) markers. It was conceived that noise on the asynchronous links could easily introduce spurious symbols and corrupt packets. (In practice, such problems have not been observed in existing machines given that the long, cabled links which had been envisaged on the original design were avoided in the end.)

Packet corruption could still occur though, if a chip is reset (due to local problems) while sending to its neighbours. There is also a parity bit in packets where space allows, as a crude intactness check.

Finally, there is a *timestamp* on potentially long-lived packets, intended to guard against misprogrammed routeing allowing packets to circulate in the system-wide network indefinitely. This is a simple, slowly changing phase number known by all routers and appended to packets as they are transmitted. To use this mechanism, all the routers in the system need to be synchronised, to some resolution. Synchronisation will not be perfect and, in any case, the time phase may change while a packet is *in flight*. A 2-bit Gray code is therefore used for the time phase, where a router will detect a mismatch on *both* bits and will remove the packet *before* trying to route it; this is separate from the dropping due to congestion. A packet will then time out if undelivered somewhere between one and two time phases after transmission. The time phases are set in software but envisaged to be of the order of a few milliseconds; legitimate deliveries should be completed in much less time than this.

2.2.4 Interconnection Networks

The SpiNNaker network connecting processors to routers and interconnecting routers is *asynchronous*. Much of the inspiration for this is to remove chip-wide timing closure issues since the subsystems can then be assembled without a need to consider the timing paths in great detail. The 40- or 72-bit packets are serialised into 4-bit elements, and these are transmitted serially between subsystems.

On chip, the coding scheme is a 3-of-6 Return-To-Zero (RTZ) code. Each 4-bit element is indicated by *activating* exactly three of the six data wires to indicate each symbol or 'flow control unit (flit)'. When the receiver sees the complete code, it acknowledges this by activating a returned signal. When the transmitter sees this, it can deactivate the symbol. Since there are 20 possible codes (Table 2.2), each flit carries a 4-bit binary value; some codes are omitted, which makes decoding slightly simpler. A separate signal *instead* of a data flit indicates EoP that allows framing detection.

A similar mechanism is used to convey flits between chips. Here, however, the wiring is on PCB tracks and any switching is quite expensive in power consumption. This is therefore minimised by using a more complex 2-of-7 Non-Return-to-Zero (NRZ) scheme (Table 2.2). This requires an extra wire per link per direction but only two wires switch for each flit and information is carried by the *transition* rather than the level, so there are two data and one acknowledge transitions per flit rather than the six data and two acknowledge transitions in the internal code (Figure 2.11). The encoding and, especially, the decoding are more

Table 2.2. Inter-chip flit encoding.

Hexadecimal	3-of-6 code	2-of-7 code
0	11_0001	001_0001
1	10_0011	001_0010
2	10_0101	001_0100
3	10_1001	001_1000
4	01_0011	010_0001
5	11_0010	010_0010
6	10_0110	010_0100
7	10_1010	010_1000
8	01_0101	100_0001
9	01_0110	100_0010
A	11_0100	100_0100
B	10_1100	100_1000
C	01_1001	000_0011
D	01_1010	000_0110
E	01_1100	000_1100
F	11_1000	000_1001
EoP	–	110_0000
–	00_0111	000_0101
–	00_1011	000_1010
–	00_1101	011_0000
–	00_1110	101_0000

complicated however! In this case, to reduce the wiring (and pin) overhead, EoP is coded as another flit. A 2-of-7 code has 21 separate symbols, so the required 17 fit comfortably [225].

The asynchronous handshake protocol relies on transmitter and receiver alternating in action. This functions well in the absence of faults but there can be a problem if one end of the communications loses state. This can happen, for example, if a chip crashes badly and takes a complete watchdog reset. Was the chip in an *active* or *passive* phase on each of its links? The solution employed is to assume

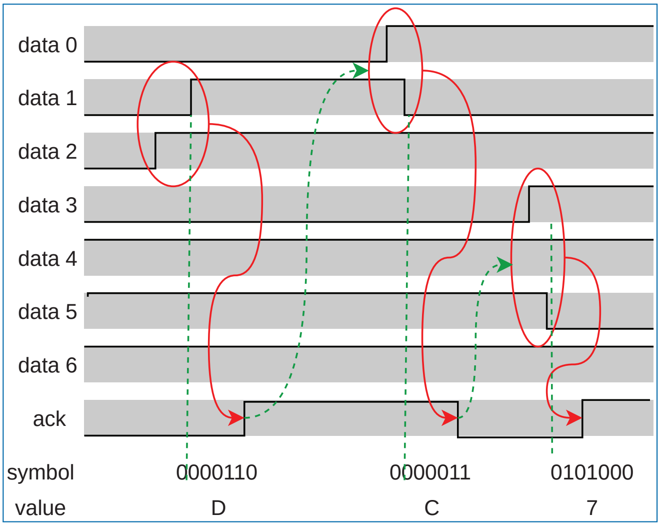


Figure 2.11. Inter-chip flit encoding: 2-of-7 asynchronous NRZ handshaking.

that the chip is active, so it can send data (as soon as it has some to send) but it also acknowledges data which it may or may not have been sent. The transition detectors will ignore a *second* transition if they are already active, so if the acknowledgement is spurious it is ignored and lost; however, if the corresponding device had just sent a flit, it is now acknowledged even though its content has been lost. Flit-level communication is resumed; the flits, including EoP markers, are forwarded to the next router which will detect an incomplete packet, discard it, raise an interrupt and resynchronise.

The network described above, the *comms* NoC, supports the SpiNNaker (short) packet communications across the entire machine. There is a second, independent network on each chip, the *system* NoC, which acts as the local shared bus. This employs the same asynchronous interconnection technology to simplify timing closure but the interface and traffic patterns are different and the topology reflects this to some extent.

The local shared resources comprise the SDRAM and *all the rest*, the latter category being peripheral interfaces et alia. The heaviest data traffic was anticipated to be to the SDRAM. The system NoC is therefore decoded near each source and crossbar-switched into these two branches, where various requests are then arbitrated and serialised. There are 19 masters on this network: the 18 processor subsystems and the router, which can read and write to shared resources, prompted by NN packets from a neighbouring device. This latter facility provides a debugging aid and allows code – and even router network tables – to be promulgated during boot.

The heaviest traffic on the system NoC is DMA from – and, to a lesser extent, to – the SDRAMs. This comprises bursts of contiguous data that are well suited to SDRAM efficiency. On the one hand, the interface to this part of the network uses an AXI interface, which is optimised for such *trains* of data and, in this case, is 64 bits wide. On the other hand, the remaining shared devices are slaved on an AHB. The system NoC bridges these different protocols.

In a similar fashion to the inter-chip links, this asynchronous interconnection can be disrupted by *unusual* events. The only anticipated problem stems from the loss of coherency due to a processor being reset *during* an outstanding transaction. The ARM itself provides no alternative but a straightforward restart; under any conditions but a full power-up, the bus bridge retains some state and is able to complete (and discard) any outstanding transactions before reconnecting the processor. This avoids a crash-reset jamming the whole network and allows the affected processor to recover. The mechanism extends to freeing up any bus locking in the unlikely event of resetting during a read-modify-write operation.

2.2.5 The Rest of the Chip

Although each subsystem is built as a conventional, synchronous unit, there is no particular frequency or phase relationship between the units. Because there is a limited number of clock sources, typically all of the processors are run at the same frequency but they will be at different phases; the router is controlled separately, according to the expected traffic and usually does not need to be run at its full design speed, while the SDRAM can be run as fast as possible. A fringe benefit of assembling the chip from many desynchronised components is that the demands on the power supplies are not correlated, reducing the high-frequency variation of supply current and making power supply decoupling simpler; the radiated electromagnetic noise is also reduced. A little effort was put in to ensure that the processors are likely to all be *out* of phase with each other.

There are numerous shared resources on the SpiNNaker chip, many of which are fairly conventional in nature. Examples are a ROM that contains initial boot code, a small SRAM used for interprocessor communication and hardware-supported semaphores for synchronisation and mutual exclusion. Additionally, shared peripheral interrupts are broadcast to all processors.

Although accessible by any processor, the intended convention is that most shared peripheral devices are only used by the selected *monitor*. For example, there are clock generators that are made as a global resource to reduce the power and area overheads. Two independent Phase-Locked Loops (PLLs) multiply an input (10 MHz) reference frequency and their outputs can be divided and switched to feed different subsystems. Processor clocks are limited to two groups (9 processors in each) but typically use the same source; 200 MHz is the design maximum frequency. The router is typically run slower because it can (but need not) be in spiking applications, and 133 MHz is convenient. The SDRAM controller is optimised separately to get better performance from the SDRAM device, and a 130 MHz clock is usually used. At these speeds, under reasonable load, the power consumption of the chip is around 1 W.

One slightly unusual *shared subsystem* is the mechanism for picking one processor to be *chip monitor*. One processor is normally dedicated to functions such as setting up and maintaining routeing tables and host communication, and this is set into the architecture as the receiver of P2P packets. Rather than dedicating a particular subsystem to this task, the selection is left to run-time. The reasoning for this is partly in consideration of increasing useful chip manufacturing yield.

Due to defects, not all manufactured integrated circuits work. Defects tend to be randomly situated, so in a chip like SpiNNaker any particular defect is likely to be in one of the processor subsystems – and, in particular, probably in its SRAM. On boot up, each processor system runs some simple tests and, if it completes these successfully, assumes that it is okay and attempts to claim the title of *monitor*. This is done by reading a particular peripheral device (in the *System Controller*)[1] which has been cleared by power-up reset. The first device to do this is granted permission to go ahead and its identity is recorded; subsequent devices are rejected and their software moves to a subservient role. If everything is still functional, the victorious monitor then brings up the whole chip.

However, the tests in the boot ROM are reasonably primitive as it was perceived as risky to commit to having too much unfixable code on the chip mask and it was envisaged that the subsequent discovery of a fault could then be fatal to the whole device. To protect against this, a second reset – such as a watchdog – will repeat the process, the difference being that the previous monitor will be refused even if, as is likely, it is still the first to ask. To have *two* subtly broken claimants to the 'monitorship' would be particularly unlikely.

As a final line of defence against faults, each chip has a hardware watchdog unit. This is intended to provide protection for the chip *monitor* which can then provide more sophisticated monitoring of the applications processors in software. It acts to reset the monitor processor after a preprogrammed interval unless itself periodically reset by the software. The unit also has a second time-out interval and a further output which will only *trip* if the monitor has not recovered after the first watchdog;

[1] The System Controller also includes functions such as individual core resets and interrupts, and sempahore registers.

this is set up to reset (and thus reboot) the whole chip, although it is anticipated that simply recovering the monitor will normally be sufficient to initiate recovery. The monitor (or, indeed, any processor) can reset any processor(s) using the *System Controller*, which can provide a reset *pulse* such that a processor can safely reset itself, if desired.

As part of the support for larger systems, there was an (inexpert) attempt to build a thermometer on each chip. This is possible because the properties of the electronic components – particularly speed – change with the temperature. Unfortunately, the properties also change with variations in (local) operating supply voltage and individual manufacturing conditions. To overcome this, three different temperature-sensitive circuits were implemented. One is a simple inverter ring oscillator, which can be timed against a known, crystal-regulated delay; the second is a mixed-signal ring oscillator whose stage delay reduces rather than increases with rising temperature; the third was a timer of the delay of the leakage discharge of a capacitor. By taking three measurements with three unknowns, it is, in principle, possible to extract values for all three, independently.

2.3 Multiprocessor Support

With any multiprocessor system, there can be a fundamental problem of synchronisation and mutual exclusion. This is not a major issue for the intended SpiNNaker applications, since these are planned to use independent processor systems interacting via (spike) message passing where the exact ordering is not important. Nevertheless, not including some means of synchronisation was liable to be a cause for future regret as it is difficult to retrofit.

Two mechanisms were provided for processor synchronisation: the first is simply the support for the ARM9's *SWP* ('swap') instruction – a single instruction which attempts a locked read-write operation on a selected byte or word in the address space. In SpiNNaker, the only meaningful addresses are in the shared memory, which is attached via an asynchronous NoC; the NoC therefore supports temporary locking of its arbiters to make the read-write operations indivisible. (Additional logic in the NoC interfaces guards against problems if the operation is aborted, such as via an embarrassingly-timed watchdog reset.)

A second mechanism, somewhat faster and more convenient, is built into the shared hardware in the *System Controller*: here a set of word addresses are mapped onto registers that are read-sensitive; two addresses are associated with each register, where reading one of the pair will set an associated Boolean flag and the other will clear it. The flag value *before* the operation is returned, thus providing indivisible test-and-set and test-and-clear operations. Thirty-two such flags are provided; if this proves inadequate, they can be used to lock larger structures or the SWP approach can be used.

2.4 Event-Driven Operation

SpiNNaker is intended to operate in an entirely event-driven fashion to optimise performance and energy consumption. There is no conventional operating system running on the cores, simply a low-level interrupt service provider kernel. A core is normally in low-power sleep mode. When an interrupt arrives, the core wakes up to process any required data, possibly requesting DMA transfers or emitting packets. On completion of the interrupt service, the core returns to sleep.

Table 2.3 lists the different sources that can interrupt the cores. The large number of interrupt sources supports efficient interrupt-driven operation as it relieves the core from wasting clock cycles having to identify the source after being interrupted.

Table 2.3. SpiNNaker chip interrupt sources.

Processor subsystem local
Communication controller packet reception (5 ×)
Communication controller flow control (3 ×)
Communication controller errors (3 ×)
DMA transfer completion
DMA transfer error (2 ×)
Local timer/counter (2 ×)
Software interrupt
Global
Router diagnostics event
Router packet error
Router un-routable packet
Watchdog timer
Slow system clock
Ethernet controller (3 ×)
On-chip inter-processor interrupt
Off-chip interrupts (4 ×)

2.5 Chip I/O

Nearly all the pins on SpiNNaker fall into one of two categories: the inter-chip links – where each of the six links comprises a complementary pair of asynchronous links with seven data and one acknowledge wire each – and the SDRAM interface. Provision was made for two 1 Gb low-power Double Data Rate (DDR) SDRAM chips (the contemporary technology) in the architecture; one such die is physically stacked onto the ASIC die and wire bonded before packaging – this makes the overall system PCB *footprint* significantly smaller – but the interface is also pinned out for expansion; in practice, extending the SDRAM has not proved necessary. These interfaces leave little room for *conventional* interfacing, but some of the remaining pins provide for this. Chiefly, the ASIC includes an Ethernet interface providing a Media-Independent Interface (MII) which provides for a host link to a standard network. This requires an external *Physical Layer Device (PHY)* – a physical medium adaptor – but it was never planned to provide Ethernet connectivity to *all* the devices, just specific selected ones to provide *gateways* to the SpiNNaker network.

The other general purpose I/O is a standard, parallel I/O port. In some cases, the bits here may be used to support (for example) the Ethernet control. One bit is read at boot time to select one of two boot options in the internal ROM: the conventional start-up and a (tested, but not generally needed or used) option to use other pins as a serial bus (Serial Peripheral Interface [SPI]) to download a different boot sequence from an external source. This second option was to guard against a serious mistake in the main boot ROM code; it has not been needed. However, some devices still use an external ROM, a good example being an Ethernet-expanded chip which needs individual data such as a Media Access Control (MAC) address. There are still several always-uncommitted bits that are useful primarily for debugging purposes and the all-important *blinkenlight*.

Finally, an IEEE 1149.1-compliant Joint Test Action Group (JTAG) port is also available for debugging purposes. Internally, the device chain comprises only the 18 ARM processors, as JTAG support was not deemed cost-effective for other system components.

2.6 Monitoring

To facilitate tuning of the system and to give feedback on the design – this is, after all, a research project – some hardware monitors were built in as counters. Fundamentally, these are used to help observe the behaviour of the communication networks.

The router has sixteen 32-bit counters which count packets in particular classes. Each counter has an input filter which can be set to include or exclude packets of a given type (such as *multicast only*), whether they have a payload, if they have been actively (as opposed to *default*) routed, where they have been routed to and so on. These are encoded as Boolean switches so a user can enable various combinations, including *all*. (Because there are so many possible internal destinations, internally the last mentioned category is divided only into each chip-to-chip link, monitor processor, any application processor and *dumped*.) Emergency routeing states are included so any re-routeing, which would otherwise be invisible, can be detected. These allow traffic patterns to be observed over time and any *hot spots* detected.

As a bonus, the filters can be used to activate interrupts, so the passing of a particular sort of packet can attract immediate attention from the monitor (or other) processor.

A second counter set monitors the behaviour of the system NoC; in this case, rather than count transactions (which are already known), it *times* the latency of a request to reveal how well the SDRAM is serving. Here counters are incremented according to the number of clock cycles between the memory request and response, which encompasses the travel time across the asynchronous network, any delays due to arbitration and the latency of the SDRAM itself. The counters are in *bins* of adjacent values and results are presented in the form of a hardware histogram, accumulated over a set period.

2.7 Chip Details

The completed ASIC (Figure 2.12) measures about 10 mm^2 and contains about 100 million transistors, mostly as static RAM. Its *feature size* is 130 nm. The processors and router can run (within specification) at 200 MHz; the processor subsystems are typically run at this frequency although it is normal to run the router at 133 MHz since it still meets its usual demands at this speed. The SDRAM interface is usually set to 130 MHz.

The power consumption depends on the active loading; the processor can halt when there is no work to do, which reduces the power consumption significantly. However with all 18 processors running at 200 MHz, the power dissipation is still around the 1 W mark.

The implementation of the SpiNNaker chip was a big challenge given the size and complexity of the system. SpiNNaker integrates several external IP devices, such as the ARM processors and SDRAM controller, with components developed in-house by the SpiNNaker team.

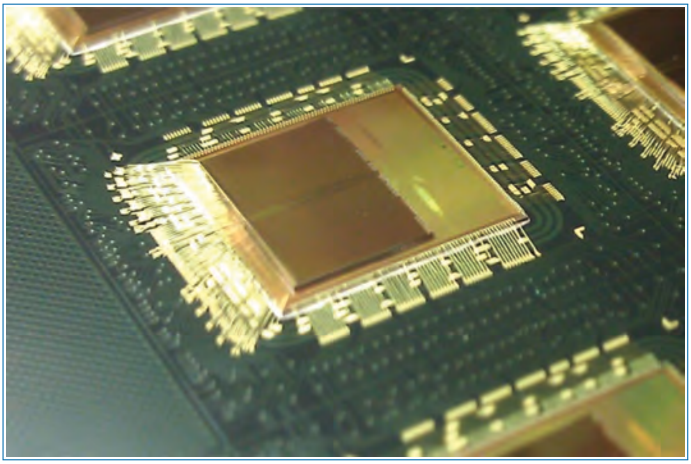


Figure 2.12. The SpiNNaker ASIC, bonded to its 'piggy-back' SDRAM. Photo courtesy of Unisem Europe, Ltd.

SystemC was used to validate the architecture and design of the SpiNNaker chip. The synchronous models were cycle accurate, while asynchronous network models were based on early delay estimates. External synchronous IP was delivered in RTL Verilog, which was also used to develop most of the in-house designs, whereas asynchronous IP was delivered in technology-mapped, gate-level Verilog. Equivalence checking was used to verify RTL synthesis and optimisation. Gate-level models with extracted parasitics and annotated delays were used for simulations.

The Synopsys Galaxy Design Platform was used for the design and implementation tasks. The implementation employs architecture and logic-level clock gating. The design methodology was fine-tuned with special emphasis on the power efficiency of the clock networks. Power-aware synthesis was used throughout the flow. A hierarchical methodology was employed [195] for the implementation of the fully asynchronous networks, encapsulating small sections of the logic in customised macros and using these as blocks for the larger sections.

The SpiNNaker chip is packaged in a 300LBGA package with 1 mm ball pitch. All IO is assumed to operate at 1.8 V with CMOS logic levels. The package exports an SDRAM interface that operates at 1.8 V LVCMOS that is usually left unconnected, as the package incorporates an internal SDRAM die.

2.8 Design Critique

There are a few, minor niggles but the ASIC is basically fully functional. At the time of writing, the SpiNNaker chips have been in use for some years. Largely they have proved to fulfil their intended function although some shortfalls have become evident. In addition, as might be expected in a research machine, the requirements have evolved and the chip cannot satisfy all of these efficiently.

Taking each subsystem in turn, the processor units are basically okay. The original target of 1000 neurons per core has proved optimistic, partly because the desired neuron models have become more complex,[2] as have the synapse models, and the number of synapses per neuron can also be higher than the original target. Depending on the models used, up to 256 neurons per subsystem is proving tractable, and currently, this is the maximum number of neurons per core supported by the software. Memory can also be a limiting factor. As something close to two-thirds of the processor subsystem's area is RAM as it is, a better way of thinking about the device is as a set of RAMs with attached processors, rather than the other way around. In this view, the RAM limits the number of neurons and synapses in each subsystem: the alternative would be to have larger RAMs by reducing the number of processors.

The original intention was to have more processing capacity than was needed, so processors could 'sleep' for a significant proportion of the time. In practice – again, partly due to increasing ambition – the processors are active for most of the time if natural, biological speeds are modelled at a 1:1 ratio. The major inadequacy in the processor now turns out to be the lack of floating-point support: some algorithms apparently *need* floating-point calculations and performing these in software is expensive.

Because the number of neurons is usually lower than the original design, the DMA units need to move less data than was expected. They have not provided any bottleneck in use, although the rate of transfer from the SDRAM is throttled when more than three or four transfers (from three or four subsystems) are active simultaneously. Use of DMA keeps this in the background and it seems to function well.

In a large machine, there were concerns for the reliability. The designers attempted to protect against radiation-induced *soft* memory errors by the inclusion of block CRCs as error detection on DMA transfers. This should at least detect lost bits in the SDRAMs. In practice, despite extensive memory tests on a large machine, no such error has yet been provably observed. On the other hand, several soft errors have been observed in the SRAMs over the same tests.[3] The SRAMs have no hardware protection mechanisms, so recovery relies on either software checking (data) or defensive software and watchdogs (code), both of which are costly. The

[2] A trend which may continue – currently, we focus on point neuron models, but interest is growing in two-compartment and dendritic computation abstractions.

[3] We are grateful to Prof. Tobias Noll for drawing our attention to this matter and to public data from Micron, with which our measured error rates are broadly consistent, at: https://www.hotchips.org/wp-content/uploads/hc_archives/hc16/1_Sun/10_HC16_pmTut_1_bw.pdf

error rates are small – negligible in a small system (say, a few thousand processors) or short time-scales but are a concern for a million processor machine running over several days.

The router copes well with the design loads and is typically run satisfactorily at half its maximum speed. With spiking neural models, the spike packets are approximately evenly distributed in time (as a result of careful software design to ensure that this is the case), so network congestion is fairly unusual. However, other users have implemented other applications – neural and otherwise – on the machine, some of which are synchronous, resulting in network idle periods interspersed with floods of packets. In these conditions, points in the network congest and, once a router is blocked, the back-pressure causes the congestion to spread. The time-out/packet drop will free this in time but it is not helped by the relatively slow (software mediated) packet dropping rates. More elasticity or faster dropping would alleviate many of the problems posed by these applications. However, *permanently* dropping packets – even neural spikes which might be expected to be fairly unimportant in a fault tolerant system – turns out to be unacceptable to many users, so the hardware dropping rates are typically limited to the speed that software can salvage all dropped packets.

The *size* of the multicast routeing tables was set by (somewhat inspired) guesswork. The number of entries here is arbitrary but sets the size of the TCAM which, in turn, dominates the router area; 1024 entries were implemented; this allows functional placement and routeing of most neural networks so far tried, although, even with some clever exploitation of the bit-fields and prioritisation of entries, it is uncomfortably small in some circumstances. Furthermore, tightly optimising the initial setting of the TCAM, such as the sharing of entries, makes subsequent, run-time modification more difficult. Neural interconnection updates primarily involve changing synaptic weights, but if new neural *projections* appear – and, in biology, they do – then changes to the TCAM may be needed to model this. A larger table would ease this process considerably. However, the existing table is not vastly too small: something like a doubling in size should easily accommodate anything so far envisioned.

The most serious architectural drawback in SpiNNaker is probably not connected with simulating spiking neurons as with being a computer. The short communication packets provide well as models of neural spikes but provide poorly for copying bytes between computer memories. There are two readily identifiable areas where this is needed:

Loading data: Code is necessarily short and is typically the same in most processors; the data tables that define the neural net are all different and are large. These tables are not hand-generated at the neuron level; they are specified statistically in *populations*. However, for convenience in the early systems, the expansion to neurons has been performed on the host server, requiring the subsequent uploading of vast data tables on relatively low-bandwidth connections.

This problem can, of course, be alleviated by loading generator functions instead and performing the data expansion within the machine.

Unloading results: Having run a simulation, the user needs to see what happened. This is the reverse problem from the data loading. Again it could be alleviated by performing some of the statistical compression in parallel in the SpiNNaker machine; currently, it involves dumping large quantities of data through the network and processing this on the host computer.

While both these problems can largely be mitigated by more sophisticated software, better (faster) up- and down-loading of the SpiNNaker's SDRAM contents would give a more generally usable machine.

The NN packets provide a means of remote access to the shared resources of a chip without the need for software cooperation from the target chip. This proves to be a valuable debug and, especially, diagnostic tool. It would be even more valuable if it could also reach into the individual TCMs of crashed processors. This was not done as it would have been a significant additional feature to make the system NoC *bidirectional* and provide a second master to the DMAC bus; however, in hindsight, it may have paid off to do this.

The purely on-chip (*system*) network, which is used primarily to DMA SDRAM contents to (and from) the individual processors' SRAMs, meets its requirements well. It does not provide enough bandwidth for a single processor to use the full SDRAM bandwidth, but this was never the intention as the SDRAM is a shared resource. In practice, three or four processors can share the SDRAM relatively unimpeded; with more active processors, the share is limited and each participant is allocated a roughly equal share. Since the applications do not require continuous SDRAM access and requests are not correlated, this network and the RAM are not a bottleneck.

One omission which was not obvious in the original design is the lack of hardware memory protection. Various hardware systems that *need* security are protected from user-mode accesses but there is no protection of the RAM itself. The reasoning was that the applications are *embedded*, the users are trusted, and therefore, the hardware overhead is unnecessary. While this reasoning still holds, the software development process has shown that the addition of some protection of the RAM would facilitate easier code debugging by helping to localise programming faults. There is an ARM standard Memory Protection Unit (MPU), which offers access control of programmed regions of the address space with little hardware overhead. With hindsight, this would probably have been a worthwhile inclusion.

The triangular connectivity of the network was partially determined by the desire to provide *emergency* routeing around broken or blocked inter-chip links; packets can be routed around a breakage via the other two sides of the triangle. In practice, this has never really been an issue; blockages are most likely due to congestion at the destination router, so finding an alternative path is not useful. This feature is therefore somewhat redundant.

Emergency routeing is unlikely to be used and, since this is the only constraint requiring the 2D triangular mesh, there are possibilities to use the chip in other network topologies. The most obvious such is a 3D cubic mesh, which can still exploit the *default routeing* feature to save on TCAM entries. A machine configured as a 3D torus has an advantage in shortening the average path length. This has not been put into practice though, since the network capacity of the machine is more than adequate for neural simulation using the original layout and the inter-chip and inter-PCB connections are already well understood – and, probably, somewhat more tractable.

Each ARM9 processor is supported by a Vectored Interrupt Controller (VIC) with 32 interrupt inputs; the particular VIC allows interrupt prioritisation, which supports nesting of interrupt service routines and half of the inputs to be vectored directly using their specific service routines; the other 16 processors need some form of software dispatcher. The choice of interrupt signals seemed fairly clear at design time as there were about 32 hardware status signals that could sensibly be used; it was largely a matter of filling up the available interrupt inputs with status signals. Only one bit was allocated as a software-triggered input, allowing software on one processor to request attention from any other.

This is typically restricted to communications to and from the *monitor* processor. The inefficiency is that a single interrupt has to serve *all* the potential communication needs, which implies software checking of status previously implanted in the (slow) shared memory. This is a particular burden for the monitor processor which needs to determine which other processor(s) are requesting attention and the reason(s) in each case by working through a collection of *flags* planted in shared RAM. In retrospect, combining some of the less important hardware signals could have made room for more software signalling which could relieve some of this burden. *Ideally*, extending the interrupt structure to cascade *more* discrete software signals would have been even more useful. This could facilitate simpler and faster message communication, particularly as all host to application processor links use P2P packets and, necessarily, are mediated by the *monitor* processor. Peer-to-peer signalling in the applications processors could also be useful in extending the flexibility of the chip running not-spiking-neuron tasks.

The temperature sensors were tested and functioned basically as predicted. Some curves have been plotted where properties could be controlled. Unfortunately (to date), the calibration and extraction of the true temperature has defeated everyone who has tried.

The asynchronous inter-chip links have proved reliable, delivering 250 Mb/s consistently; a modest speed by current standards but, as prioritised, extremely energy efficient. The links scale more than adequately to massive sizes: the full-size SpiNNaker system, described in Chapter 3, contains over 57,000 SpiNNaker chips with a bisection bandwidth of 480 Gb/s and a worst-case latency in the 34–46 μs range.

2.9 Summary

The SpiNNaker chip was designed by a small team of academic researchers and postgraduate students with the associated restrictions and constraints regarding fabrication cost and access to process technologies, standard cell libraries and intellectual property. Overall, a 40 person-years effort was devoted to its design, implementation and verification. A test chip with two processors was taped out in August 2009 followed by the production chip in December 2010. Key SpiNNaker figures are listed in Table 2.4.

Although SpiNNaker is a high-performance architecture highly optimised for running neuroscience applications, it can also be used for other distributed computing, such as ray tracing and protein folding. The chip provides a cost-effective means of achieving over 10^{14} operations per second, provided that floating-point arithmetic is not required.

As a message-passing system, the greatest performance bottleneck is the communications between processors and, therefore, SpiNNaker was optimised for the short, multicast messages (spikes) associated with neural network simulation. This optimisation has resulted in some additional overhead for other applications, including loading and control of neural networks, which is done by sharing the run-time network. In hindsight, provision for larger payloads with guaranteed delivery would relieve the software burden in sharing these disparate tasks. However, the decision to share a single network still appears sensible, and this relieves some of the system-level problems.

Experimental results show that, for massively parallel neural network simulations, the customised multi-core architecture is energy efficient while keeping the flexibility of software-implemented neuronal and synaptic models, absent in current neuromorphic hardware.

Table 2.4. SpiNNaker chip key figures.

Process	
Process technology	UMC 130 nm 1P8M CMOS
Die area	101.64 mm^2
Power supply	1.2 V (Core), 1.8 V (I/O)
Processing	
Processor cores	18 ARM968s (1 monitor, 17 application)
Processor frequency	200 MHz
Processor node area	3.75 mm^2
Memory	
Local memory per core	32 KByte (instruction), 64 KByte (data)
On-die shared RAM	32 KByte
Off-die shared RAM	128 MByte DDR2 SDRAM
Communications	
On-chip interconnect	Asynchronous NoCs
Off-chip link b/w	250 Mb/s
On-chip comms Link b/w	5.0 Gb/s
Off-die SDRAM b/w (DMA)	7.2 Gb/s
On-die shared RAM b/w (DMA)	3.2 Gb/s
On-die shared RAM b/w (bridge)	200 Mb/s
Router input b/w	5.3 Gb/s
Full-size system bisection b/w	480 Gb/s
Power Consumption	
Peak (chip)	1 W
Idle (chip)	360 mW
Idle (core)	20 mW
Off-chip link (full speed)	6.3 mW (25 pJ/bit)
SDRAM	170 mW
Implementation	
Transistor count	$\sim$100 million
Design effort	40 person-years

DOI: 10.1561/9781680836530.ch3

Chapter 3

Building SpiNNaker Machines

By Luis A. Plana, Steve Temple, Jonathan Heathcote, Dave Clark, Jeffrey Pepper, Jim Garside and Steve Furber

Strive for perfection in everything you do. Take the best that exists and make it better. When it does not exist, design it.

— Sir Henry Royce

The 40 *person-year* effort required to develop the SpiNNaker chip constituted only the first step in the path to build a platform to help understand how the human brain works. The next step was to make SpiNNaker chips work together in a massive scale to simulate very large spiking neural networks. The initial target was to simulate a billion neurons, around 1% of the human brain, in real time. Our estimates suggested that it would require one million processing cores, that is, over 57,000 SpiNNaker chips.

Figure 3.1 illustrates the road to a billion neurons. The monumental task of assembling the million-core, massively parallel *SpiNNaker1M* computer would involve configuring, testing and deploying 1,200 SpiNNaker boards, 150 power supplies, 60 network switches, 50 fan trays and 1.5 km of high-speed interconnect cables, all housed in 10 standard 19" cabinets.

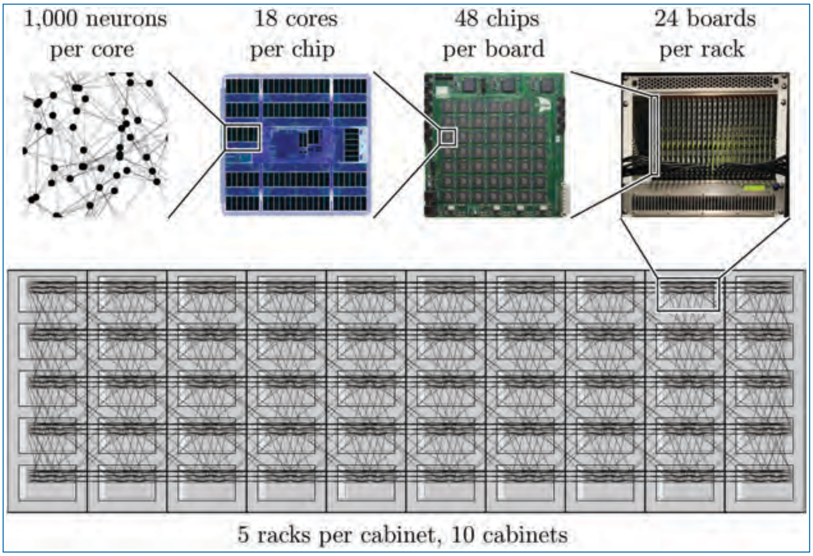


Figure 3.1. SpiNNaker machine to simulate a billion neurons in real time.

3.1 Putting Chips Together

At the time that this book was written, there were around 300 SpiNNaker boards in use in many places around the world, as shown in Figure 3.2. Additionally, 1,200 SpiNNaker boards were used to deploy *SpiNNaker1M*, located at the University of Manchester. On the way to the commissioning of this machine, five

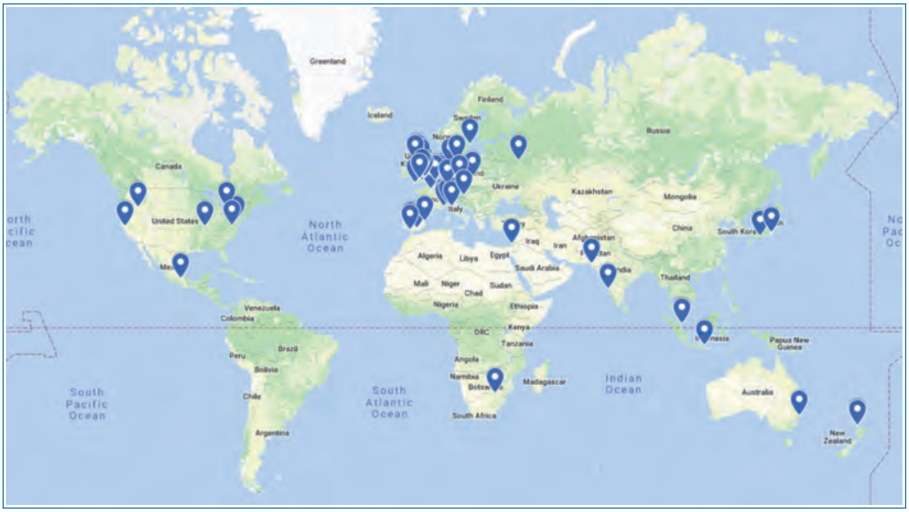


Figure 3.2. SpiNNaker boards around the world.

Table 3.1. SpiNNaker boards.

Board	Function	Devices	Notes
SpiNN-1	Technology de-risking	4× test chip + SDRAM	Internal use
SpiNN-2	Chip test	4× SpiNNaker + SDRAM	Internal use
SpiNN-3	Development platform	4× SpiNNaker	150 produced
SpiNN-4	Production prototype	48× SpiNNaker	V supply issues
SpiNN-5	Machine production	48× SpiNNaker	1,400 produced

different SpiNNaker board designs were produced, each with its own objectives and characteristics.

Table 3.1 summarises the function and main features of each of the boards. The first two boards were used mainly to verify and evaluate some of the novel aspects of the SpiNNaker chip, such as the asynchronous NoC interconnect [196]; the SDRAM interface, which contains an asynchronous, programmable digital Delay-Locked Loop (DLL) [72]; and the asynchronous, delay-insensitive chip-to-chip interconnect [224]. These boards were fitted with Zero-Insertion-Force (ZIF) sockets to facilitate the testing of packaged chips. They also had external SDRAM chips in case the on-package, wire-bonded SDRAM failed. The latter boards do not have external SDRAM as the internal setup proved reliable.

It is worth noting that, as indicated in Table 3.1, the SpiNN-4 prototype board had power supply issues. In hindsight, it should have been obvious that 864 ARM cores *waking up* concurrently can be extremely taxing on the power supply and this requires adequate capacitance and remote voltage sensing. The SpiNN-5 production board has a completely redesigned power supply and distribution network to avoid these issues.

The SpiNN-3 development platform and SpiNN-5 production board are extensively used and are described in the following sections.

3.1.1 SpiNN-3: Development Platform

Figure 3.3(a) shows a SpiNN-3 photograph and 3.3(b) shows its system diagram. The board is a software/hardware development platform for SpiNNaker-based neuromorphic systems. There are around 100 SpiNN-3 boards deployed around the world. The board houses 4 SpiNNaker chips that contain 72 ARM cores in total. A small serial ROM stores non-volatile information such as the Internet Protocol (IP) and MAC addresses, and the core and channel blacklists discussed in Section 3.1.3 below.

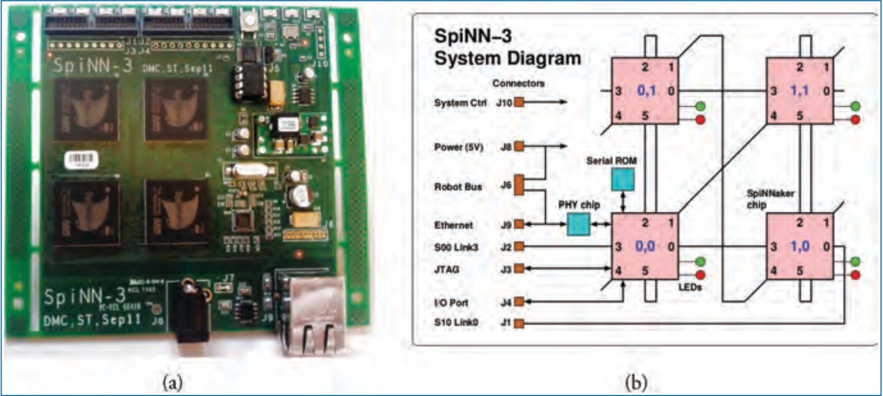


Figure 3.3. SpiNN-3: SpiNNaker development board.

Table 3.2 summarises the SpiNN-3 board main features. Each SpiNNaker chip has a red and a green Light-Emitting Diode (LED) for general purpose use. Its main I/O interface is a 100 Mb/s Ethernet connection, that is usually connected to a host machine, as described in Chapter 4. Additionally, two inter-chip SpiNNaker channels have been exported to connectors and can be used to connect to other SpiNNaker boards or to external neuromorphic devices, such as a Dynamic Vision Sensor (DVS) [141, 143, 219], also known as a silicon retina or an event camera.

SpiNN-3 boards have been used during the design, verification and testing of the different software components described in Chapter 4 as well as in the training of SpiNNaker users. Additionally, the SpiNN-3 platform has been used to build exemplar neuromorphic systems. Figure 3.4 shows a real-time, event-driven neuromorphic system for goal-directed attentional selection developed by Galluppi *et al.* [67]. The system uses a Field-Programmable Gate Array (FPGA) interface board to connect an AER [136] DVS to the SpiNNaker board. The interface, built using components from the SpiNNaker I/O library (spI/O) [197], is described in SpiNNaker Application Note 8 [198].

3.1.2 SpiNN-5: Production Board

Figure 3.5 shows (a) a SpiNN-5 photograph and (b) a board diagram. SpiNN-5 is a production board used to build multi-board machines, which can be as small as 1 board and as large as 1,200 boards. The board houses 48 SpiNNaker chips that contain 864 ARM cores. A Flash Memory (Flash) stores non-volatile information such as the IP and MAC addresses, and the core and channel blacklists discussed in Section 3.1.3 below.

Table 3.3 summarises the SpiNN-5 board main features. Each SpiNNaker chip has a green LED for general purpose use. Its main I/O interface is a 100 Mb/s

Table 3.2. SpiNN-3 main features.

Feature	Value
Dimensions	90 mm × 100 mm × 1.6 mm
SpiNNaker devices	4
PCB layers	6
PCB track/gap	0.15 mm (all signal layers)
Via hole size	0.25 mm
PCB stackup	TOP signal
	2 GND
	3 1.2 V
	4 1.8 V
	5 signal
	BOTTOM signal
External power	5 V DC
On-board power	1× 1.2 V (PTH05050W)
	1× 1.8 V (TPS77818d)
	1× 3.3 V (TPS77833d)
Interfaces	2× SpiNNaker channel
	1× 100 Mb/s Ethernet (SMSC8710A)

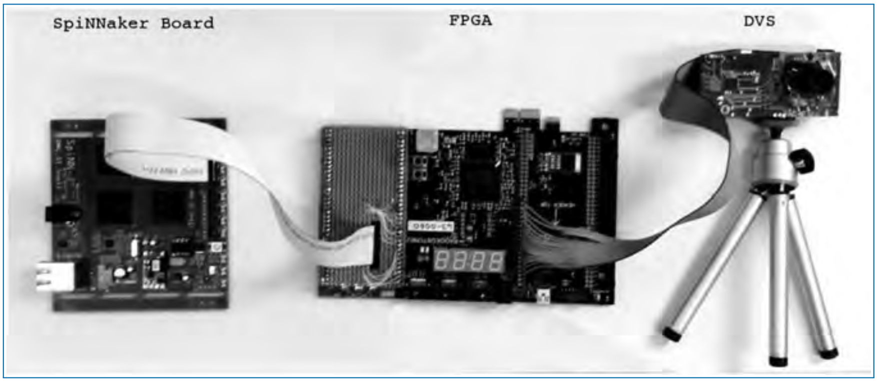


Figure 3.4. SpiNN-3: Event-driven neuromorphic system.

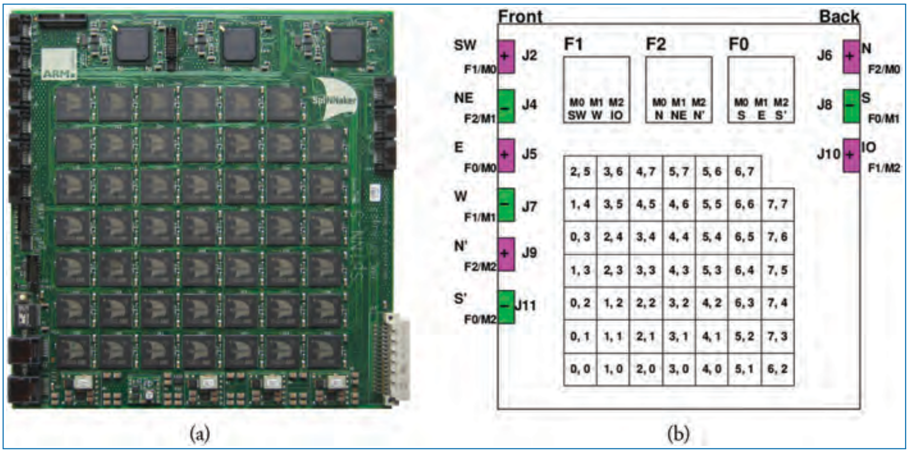


Figure 3.5. SpiNN-5: SpiNNaker production board.

Ethernet connection, that is usually connected to a host machine, as described in Chapter 4. Additionally, a single inter-chip SpiNNaker channel is exported to a connector and can be used to interface to external neuromorphic devices.

In Figure 3.5, the SpiNNaker chips appear to form a square but they are actually organised as a hexagon, as shown in Figure 3.6(a). This shape allows efficient board tiling to create multi-board machines [94]. Figure 3.6(b) shows how 12 SpiNN-5 boards can be tiled together to form a 24 × 24 chip two-dimensional ($2D$) hexagonal mesh. The (*red*, *green* and *blue*) coloured lines represent board-to-board connections. The $2D$ hexagonal mesh is converted into a hexagonal torus, the SpiNNaker machine topology, by wrapping around the edge connections, as shown in Figure 3.6(c). Some of these connections *wrap around* from the north edge to the south edge and from the west edge to the east edge. It is interesting to note that, in some cases, the *wrap around* connections are on-board SpiNNaker channels and not board-to-board connections, for example, the top half of the board labelled $B5$ is located on the south edge while the bottom half of that board is located on the north edge.

Figure 3.5 shows that the SpiNN-5 board also houses three FPGAs and nine Serial AT Attachment (SATA) connectors. The six connectors on the front edge are used to connect boards to each other and the remaining three can be used to connect peripherals such as the DVS described in Section 3.1.1 above. Details of board-to-board interconnect are given in Section 3.2 below. Finally, the board houses an additional ARM processor, the Board Management Processor (BMP). This core is not used to run SpiNNaker applications. Instead, as its name implies, it is used to control the operation of the SpiNN-5 board. The BMP is in charge of powering up and down the SpiNNaker chips, configuring the board FPGAs, controlling cooling

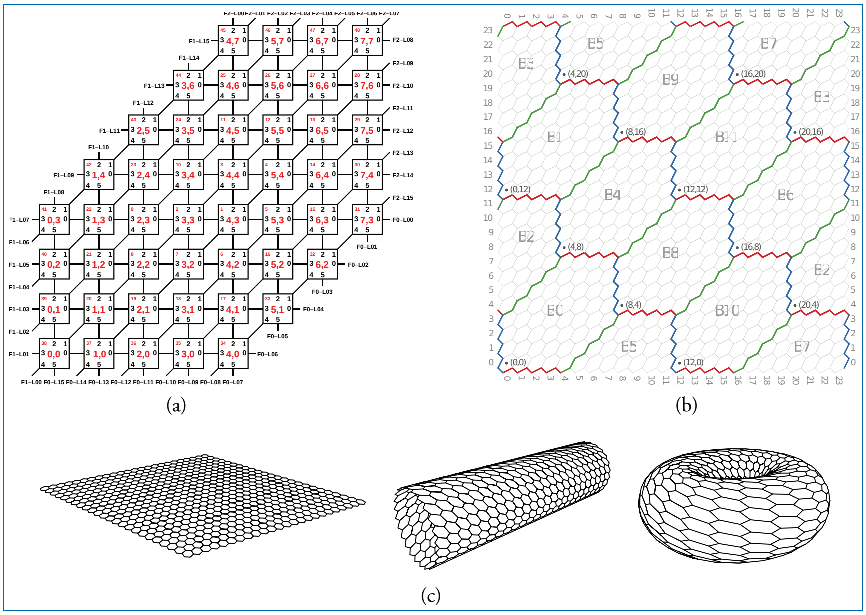
Table 3.3. SpiNN-5 main features.

Feature	Value
Dimensions	220 mm × 234 mm × 2 mm (double-height Eurocard)
SpiNNaker devices	48
FPGA devices	3× Xilinx Spartan-6 XC6SLX45T
PCB layers	12
	(*Cu* thickness, core and pre-preg adjusted to manufacturer recommendations to provide correct impedance on TOP and BOTTOM layers and mechanical rigidity during assembly)
PCB track/gap	0.125 mm/0.125 mm internal signal
	0.125 mm/0.200 mm SATA controlled impedance tracks
	0.127 mm/0.127 mm TOP and BOTTOM signal
Via hole size	0.25 mm – 25,000 vias (!)
PCB stack up	TOP signal (100 **Ω** controlled differential impedance)
	L2 GND
	L3 1.2 V a,b,c
	L4 signal
	L5 3.3 V, 2.5 V – voltage management internal planes for FPGA
	L6 signal
	L7 signal
	L8 GND, 12 V internal plane to voltage regulators
	L9 signal
	L10 1.8 V
	L11 GND
	BOTTOM signal (100 **Ω** controlled differential impedance)
External power	12 V DC
On-board power	3× 1.2 V (PTH08T221W)
	1× 1.8 V (PTH08t221)
	1× 3.3 V (PTH08t230W)

(*Continued*)

Table 3.3. SpiNN-5 main features (continued).

Feature	Value
Interfaces	1× SpiNNaker channel 2× 100 Mb/s Ethernet (LAN8710A) 9× 3.3 Gb/s SATA links – 6 on front edge, 3 on back edge 1× BMP JTAG 1× FPGA JTAG 1× DIN41612 32-way backplane connector 8× Status LEDs



(a)

(b)

(c)

Figure 3.6. SpiNN-5: Board structure and multi-board tiling. Figure (c) reproduced with permission from Heathcote [94].

fan speed, keeping track of board operating temperatures (using temperature sensors located at the north and south edges of the board) and taking appropriate action in case of overheating. The BMP has its own Ethernet connection, that can be used by the host to send commands to, and receive information from, the BMP.

The design of the SpiNNaker chip targeted energy efficiency as a top priority and the board design also reflects this goal. Table 3.4 shows the power consumption of the SpiNN-5 board under different loads. Unfortunately, due to area and design

Table 3.4. SpiNN-5 power consumption.

SpiNN-5 state	Power
SpiNNaker (idle – not booted)	28 W
SpiNNaker (idle – booted)	32 W
SpiNNaker (maximum load – SDRAM test)	70 W
FPGAs (configured – included above)	5 W

constraints, the board is not adequately instrumented to measure the power consumption of individual components, such as the SpiNNaker chips, the Ethernet and SATA interfaces or the BMP subsystem, only the power consumption of the FPGAs can be determined independently. However, board-wide measurements suggest that each SpiNNaker chip consumes around 1 W when fully loaded.

3.1.3 Nobody is Perfect: Testing and Blacklisting

Around 75,000 SpiNNaker die were fabricated and tested. Each *good* die was packaged together with an 128 MByte SDRAM die to produce a SpiNNaker chip. The chips were tested after packaging and those that reported at least 17 working cores were accepted for use in the production of SpiNNaker boards. Accepting one nonfunctional core per chip made it possible to build *SpiNNaker1M* within a reasonable research budget, given the actual manufacturing yield. As a result, every SpiNN-5 board was populated with forty 18-core and eight 17-core SpiNNaker chips.

During the chip boot process, every core runs a Power-On Self-Test (POST). A core that fails its own test indicates its unavailability through the *System Controller* (see Chapter 2). A *monitor* core is selected amongst those that pass the POST. This monitor core is responsible for testing the shared resources on the chip, such as the SDRAM and the router, and takes part in the system boot process, while the rest of the cores are in power-down mode.

Every SpiNNaker board is tested using a test suite that includes the tests listed in Table 3.5. As indicated in the table, a BMP failure is considered fatal and results in the rejection of the SpiNN-5 board. Other failures constrain the resources available but do not necessarily result in the rejection of a board. In those cases, it is vital to identify the misbehaving devices or communications channels.

During the initial board tests, it became clear that the POST coverage was insufficient. Unfortunately, the POST procedure was committed to the on-chip ROM and could not be extended or modified in any way. An alternative approach was implemented, inspired by the mechanism used on hard disks to manage bad sectors. Instead of relying on the POST, the boards are thoroughly tested offline and a list

Table 3.5. SpiNN-5 test suite.

Test	Result
BMP test	Accept/reject board
Core functionality	Identify misbehaving cores
SDRAM functionality	Identify misbehaving chips
Channel test	Identify misbehaving chip-to-chip channels
Power-cycling/reboot	Identify intermittent devices and channels

Table 3.6. SpiNN-5 blacklisting results.

Element	Blacklisted	Percentage
Total boards tested	1,333	96.95
Blacklisted chips	907	1.42
Blacklisted cores	12,148	1.07
Blacklisted channels	343	0.18

of unreliable devices and channels is kept in the non-volatile memory of the board. The *blacklist* is applied during the boot process, guaranteeing a consistent, reliable system. As explained in Chapter 4, the host reads the SpiNNaker machine information to map the application only to correctly operating devices and channels.

Table 3.6 shows the results of the blacklisting process. Entire chips were blacklisted for a number of reasons, usually involving a shared resource, such as the SDRAM or the SpiNNaker router. The number of blacklisted cores does not include the cores in the blacklisted chips but includes the cores that were already identified as not fully functional in accepted 17-core chips. Although the percentages of blacklisted chips and cores are small, they are not negligible.

3.2 Putting Boards Together

As in all supercomputers, the SpiNNaker network interconnects its processors in a topology that defines how different processors may communicate with each other. Unlike the *tree* and *N-dimensional* torus topologies found in contemporary supercomputers, SpiNNaker employs a *hexagonal torus* topology, as shown in Figure 3.6(a). In this topology, every node is connected to six neighbouring nodes, fitting together in a honeycomb-like pattern. Figure 3.6(b) shows that, due to the hexagonal arrangement of the SpiNNaker chips on the SpiNN-5 board, the hexagonal torus topology also applies when a board is considered a network node.

3.2.1 SpiNNaker Topology

The hexagonal torus topology was chosen over more conventional network topologies – such as $2D$ or $3D$ tori (sometimes known as a 2-ary N-cube or 3-ary N-cube, respectively) – due to its balance of theoretical performance and practicality. The bisection bandwidth of a topology indicates the theoretical worst-case total throughput the network is able to sustain. In networks with homogeneous link throughput, bisection bandwidth is determined by the number of links cut by a balanced bisection of the network. In an $N \times N$ $2D$ torus topology, the bisection bandwidth is $2N$ links and each node requires four links. The hexagonal torus topology requires six links per node but provides double bisection bandwidth ($4N$ links). The $3D$ torus topology also requires six links per node but by connecting the nodes differently achieves a bisection bandwidth of $8N$ links. The $3D$ torus topology, however, comes at a price – when immersed into the (approximately) $2D$ space provided by a large machine room or row of server cabinets, some connections require long cables. By contrast, the $2D$ and hexagonal torus topologies are both inherently two dimensional and consequently do not suffer from this effect. The hexagonal torus topology, therefore, shares the practicality of construction of a $2D$ torus while still gaining some of the performance of a $3D$ torus topology. In addition, because nodes in a hexagonal torus topology have a greater number of links, greater redundancy is available in the network to tolerate faults.

The hexagonal organisation of the chips is also efficient in the number of SpiNNaker channels on the board boundary. This is extremely important, given that, as shown in Figure 3.6(a), 48 channels are located at the board periphery and need to be connected to neighbouring boards. The three FPGAs shown in Figure 3.5, labelled F0, F1 and F2, are used to implement board-to-board connections. Each FPGA handles the interconnection of 16 SpiNNaker chip-to-chip channels, that is, two sides of the hexagon. Figure 3.7 shows how the channels connect to the FPGAs. Each of these channels has a bandwidth of 250 Mb/s and is used to transport short packets that usually represent neural spikes.

3.2.2 spiNNlink: High-speed Serial Board-to-Board Interconnect

As explained in Chapter 2, SpiNNaker channel data are transmitted using a delay-insensitive, 2-of-7 code. Each channel direction uses 7 wires to represent data and one additional wire as an acknowledge to complete data handshakes.

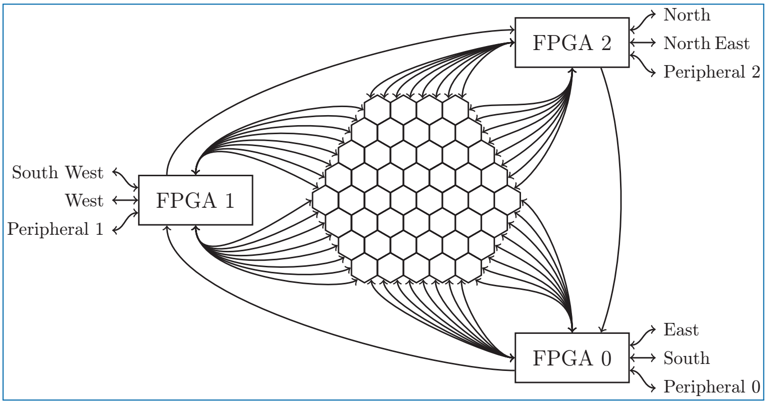


Figure 3.7. SpiNN-5: Periphery chip-to-chip channels exported through FPGAs.

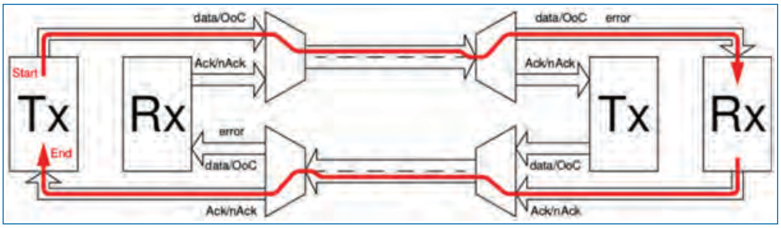


Figure 3.8. spiNNlink FPGA-based inter-board interconnect.

This energy-efficient code is ideal for inter-chip interconnect, given that it works correctly in the presence of unbounded delays. Unfortunately, the number of wires required for the 2-of-7 encoding would be extremely expensive for the direct board-to-board connection. The use of that code would amount to a total of 768 wires on the board periphery. To reduce the number of wires, the SpiNNaker channels are multiplexed over *SpiNNaker board-to-board links (spiNNlinks)*, that is, High-Speed Serial Links (HSSLs) implemented using on-board FPGAs. Each FPGA manages the 16 SpiNNaker channels on two adjacent sides of the hexagon and has spare capacity to manage peripheral connections. Additionally, the FPGAs themselves are connected in a high-speed ring, as shown in Figure 3.7.

spiNNlink incorporates several novel ideas including a bespoke, credit-based, reliable frame transport protocol that allows the multiplexing of asynchronous channels over a high-speed serial link and an efficient FPGA to asynchronous channel interface that provides twice the throughput of traditional synchronisation schemes.

Figure 3.8 shows two connected SpiNN-5 boards, each with its *transmitter* (*Tx*) and a *receiver* (*Rx*). Two independent *data* + *control* streams are multiplexed onto the same HSSL. The figure highlights one of the streams, with left-to-right flow of

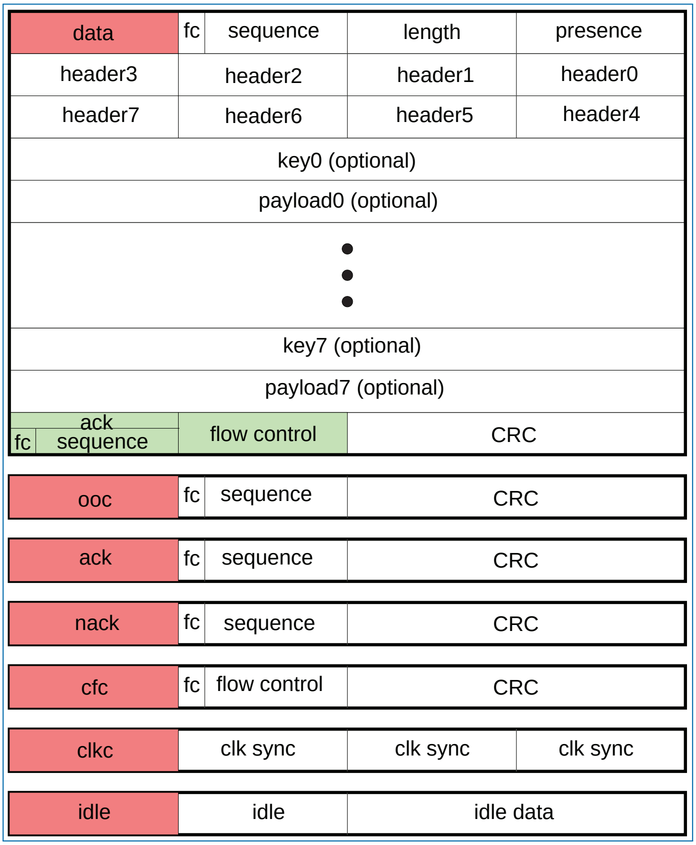


Figure 3.9. spiNNlink frame formats.

data and the corresponding right-to-left control flow. In the symmetric stream (not shown in the figure), data and control flow in the opposite directions.

Transmission over HSSLs is structured in frames. The different frame formats are shown in Figure 3.9. There are five frame types associated with data and control transmission: data (*data*), out-of-credit (*ooc*), acknowledge (*ack*), reject or negative acknowledge (*nack*), and channel flow control (*cfc*). Each frame is identified by a different *start-of-frame* special character, highlighted in red in Figure 3.9, carries a *frame colour* (*fc*) and is protected by a CRC checksum (*CRC*). *Data*, out-of-credit (*ooc*), *ack* and *nack* frames also carry a sequence number (*sequence*). Two additional frame types, clock correction (*clkc*) and idle (*idle*), are used to keep the HSSL synchronised.

Frames are a single 32-bit word long except for *data* frames, which have a variable length. As indicated earlier, eight SpiNNaker channels are multiplexed into a single HSSL and, as a result, a single *data* frame can carry up to eight SpiNNaker packets, one from each channel. A SpiNNaker packet consists of an 8-bit header, a 32-bit routeing key and an optional 32-bit payload. The 8-bit *presence* field is a bitmap used to indicate if the frame carries a packet from the respective channel. Similarly, the 8-bit *length* field is a bitmap that indicates if the packet is long (contains a payload) or short (no payload). These two bitmaps, part of the first word of every *data* frame, establish the actual structure and length of the frame. Depending on the number of SpiNNaker packets carried, *data* frames can be 4 to 20 32-bit words long.

Transmission over the HSSL operates as follows: data from *Tx* are sent once, identified by a frame sequence number and protected by a CRC for error detection. Multiple frames can be sent successively, subject to credit limits. Data frames need not contain any actual data. If the credit becomes exhausted, *Tx* simply sends unsequenced *Out-of-Credit (ooc)* frames instead.

Received data frames are either correct or not. A correct data frame will pass error checks and have the expected sequence number. Erroneous frames are rejected and retransmission is requested using the sequence number. To guarantee frames are received in order, erroneous frames also change the receiver colour so that subsequent, correct or incorrect, frames can be flushed until the erroneous frame is retransmitted correctly in the new colour.

Rx provides updates on its status to *Tx* at expedient intervals. These are not necessarily triggered by data arrival and continue in the absence of new data. Information is conveyed on the credit available, the colour and *Rx* status. Flow control information (*Xon*/*Xoff*) for individual SpiNNaker channels is also transmitted. Error tolerance is provided by the repetition of these 'frames'.

Tx re-credits its data frame allowance in response to the receiver status. Old data, retained for possible retransmission, can be discarded up to the *acknowledged* (*ack*) sequence number. When an error indication (*nack*) is received, the transmitter changes colour, ignoring further prompts until the data stream is re-established, resets its inputs to the error point and retransmits frames from the failed frame sequence point. There is no requirement that the data contained is the same as the original frames; frames may be reformed with additional data if desired.

The fully-asynchronous, handshake-based SpiNNaker channels described in Chapter 2 pose a throughput challenge for spiNNlink. In a traditional interface, the communications throughput is limited by the latency introduced by the synchronisation flip-flops required for the handshake signals.

We investigated a fully asynchronous, that is, *clockless*, version of spiNNlink [145] that used an asynchronous First In First Out (FIFO) buffer to avoid the latency penalty imposed by the synchronisers. The new design increased significantly the communication throughput but, as commercial CAD tools target synchronous design flows, also increased the design, synthesis, placement and verification effort and was not a good match for the target FPGA devices.

To avoid these issues, a novel strategy was developed for spiNNlink using well-understood synchronous timing assumptions to predict the arrival of the next SpiNNaker channel handshake, without actually waiting for it to complete. Additionally, the novel interface is aware of asynchronous back-pressure, that is, situations in which the asynchronous channel stalls for an unbounded time due to traffic congestion. In order to operate correctly in these situations, spiNNlink uses *Synchronous Timing Asynchronous Control (STAC)*. It predicts handshake timing except at the point where the channel may apply back-pressure, where it completes a fully asynchronous handshake, responding correctly to back-pressure and providing twice the communications bandwidth of the traditional implementation.

spiNNlink was implemented on the SpiNN-5 board FPGAs using Xilinx IP and the components available in the SpiNNaker I/O library (spI/O) [197]. Table 3.7 summarises the high-speed serial interconnect main features.

Table 3.7. spiNNlink main features.

Feature	Value
High-speed link bit rate	3.0 Gb/s
Board-to-board latency	665 ns
Data encoding	8 b/10 b + disparity
Tunable link parameters	Tx driver swing Tx pre-emphasis Rx equalisation
Error detection/correction	16-bit CRC + retransmission on error
Clock correction (CLKC)	CLKC code removal/duplication
SATA cable presence	automatic detection
SpiNNaker interface	Synchronous Timing Asynchronous Control (STAC)
SpiNNaker link bandwidth	5.5 Mpackets/s
Register bank	SPI interface to BMP Tx/Rx FRAME counters CREDIT status/OUT-OF-CREDIT counter CRC/FRAME/DISCONNECT error counters tunable parameter control

3.3 Putting Everything Together

As indicated earlier, SpiNN-5 production boards can be connected together to build SpiNNaker machines. Figure 3.10 shows two examples of *small-scale* machines.

A single-board machine, containing one SpiNN-5 board, is shown in Figure 3.10(a). This machine is adequate for software development and small-scale neural network simulations. Figure 3.10(b) shows a 24-board machine that consists of a fully-populated 19″ cabinet. The *black* SATA cables used for board-to-board interconnect are clearly visible on the front of the machine. Also visible are the *white* Ethernet cables for access to the SpiNNaker chips and the BMP. This machine offers over 20,000 ARM processing cores with around 1.5 KW power consumption when fully loaded.

3.3.1 *SpiNNaker1M* Assembly

Large-scale SpiNNaker machines are modular systems. The basic building block is a standard 19″ card frame that holds 24 SpiNN-5 boards and the required ancillary equipment. Figure 3.11(a) shows the front of the card frame with 24 boards installed, and Figure 3.11(b) shows the back of the card frame with three 650 W power supplies and the card frame backplane. Each module also includes a fan tray, located below the card frame, that pulls air from the front of the card frame, between the boards, and blows it to the back. Finally, the module contains a 26-port Ethernet switch that connects the SpiNN-5 boards.

The card frame backplane houses a serial ROM that contains information about the frame location in the machine. These data are used during the boot process to

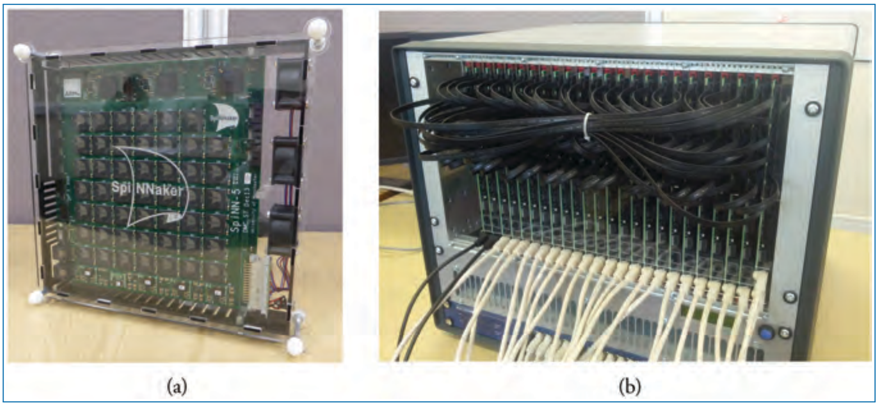


Figure 3.10. Small-scale SpiNNaker machines. (a) A cased 48-node 864-core board. (b) A 24-board 20,736-core machine.

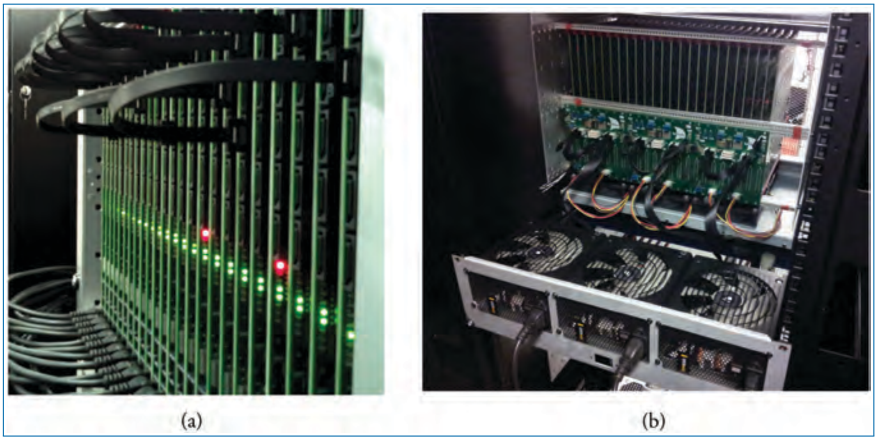


Figure 3.11. A card frame holds 24 SpiNN-5 boards, power supplies and a backplane.

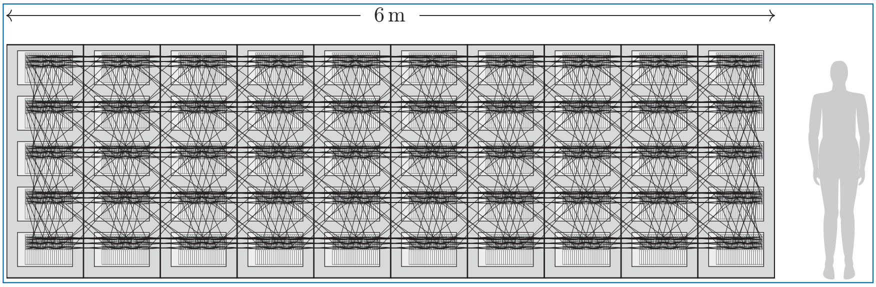


Figure 3.12. *SpiNNaker1M*: 10 cabinets and 3,600 SATA cables interconnecting them. Figure reproduced with permission from Heathcote [94].

configure the IP and MAC addresses of each SpiNN-5 board. It also contains power supply data as well as information about fan speed control and temperature limits. Additionally, the backplane provides access to temperature sensors and to a Liquid-Crystal Display (LCD) located on the front of the card frame. The display is used to provide information, such as operating temperature and power supply levels, to the machine operator. Finally, the backplane carries a Controller Area Network (CAN) bus, used by the BMP to communicate with each other. The SpiNN-5 boards are connected to the backplane through an edge connector, located at the bottom right corner in Figure 3.5(a).

To build larger machines, card frames are assembled together in 19″ cabinets. Each cabinet holds five card frames, for a total of 120 SpiNN-5 boards, containing 5,760 SpiNNaker chips/103,680 ARM cores. Figure 3.12 shows the 10 cabinets and 3,600 SATA cables required to build *SpiNNaker1M* that contains 1,036,800 cores.

Heat management is a significant design concern in massively parallel systems. Supercomputers usually consume large amounts of power, in the order of Megawatts, that are ultimately converted into heat. Most supercomputers require bespoke liquid or mixed air-liquid cooling systems. The focus on energy-efficient design and construction means that *SpiNNaker1M* consumes well under 100 KW when all cores are operating at full load, simplifying heat management. A couple of chillers blow cool air into the machine room. Air enters the machine through the cabinet front doors, and the card-frame fans move it through the SpiNN-5 boards towards chimneys located at the top of each cabinet where it is steered back to the chillers through a plenum. One of the chillers, with the associated chimneys and plenum, can be seen in Figure 3.17. The chillers transfer heat to a water system through coils. The chillers are controlled by thermostats measuring the temperature in the chimneys. Each chiller has a nominal capacity of 70 KW and a nominal air flow capacity of 3.7 m^3/s.

3.3.2 *SpiNNaker1M* Interconnect

As with any supercomputer, physically assembling a large SpiNNaker machine poses many practical challenges in terms of arranging, installing and maintaining the thousands of metres of network cables required. Cabling techniques for conventional architectures and network topologies are well understood and embodied by industry standards such as TIA-942 [250]. Unfortunately, the use of a hexagonal torus topology for SpiNNaker renders existing approaches inadequate.

Naïve arrangements of torus topologies, hexagonal or otherwise, feature physically long *wrap-around* cables that connect units at the peripheries of the system. Long connections can be problematic for several reasons:

Performance: Signal quality diminishes as cables get longer, requiring the use of slower signalling speeds, increased error correction overhead or more complex hardware.

Energy: Some energy is lost in cables; longer cables lose more signal energy requiring higher drive strengths and/or buffering to maintain signal integrity.

Cost: Shorter cables are cheaper than long ones. Longer cables imply more cabling in a given space making the task of cable installation and system maintenance more difficult, increasing labour costs by as much as 5× [40].

In some cases, long connections in supercomputers may be eliminated by creative physical organisation of the system. For example, the distinctive ‘C’-shaped design of early Cray supercomputers was chosen to reduce the lengths of physical connections and thus improve system performance [255]. Unfortunately, this approach

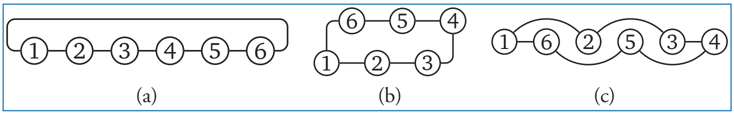


Figure 3.13. Folding and interleaving a ring network to reduce maximum cable length. Figures reproduced with permission from Heathcote [94].

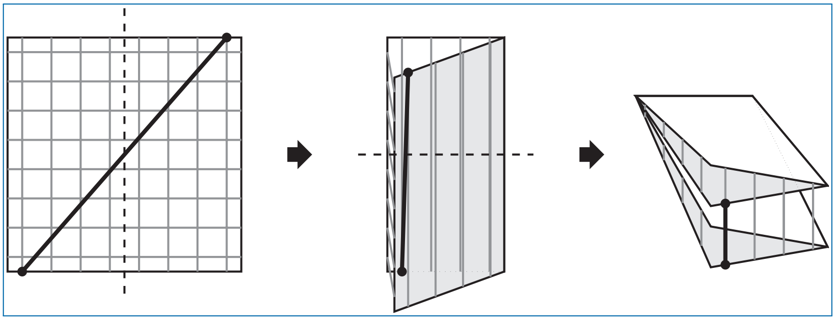


Figure 3.14. Network folding to shorten interconnect. Figures reproduced with permission from Heathcote [94].

does not scale up in the general case and requires potentially expensive bespoke physical infrastructure. Alternatively, the need for long cables is often eliminated by folding and interleaving units of the network [42]. This process is illustrated for a 1D torus topology (a ring network) in Figure 3.13. A naïve arrangement of units in this topology results in a long cable connecting the units at the ends of the ring (Figure 3.13(a)). To eliminate these long connections, half of the units are 'folded' on top of the others (Figure 3.13(b)) and then this arrangement of units is interleaved (Figure 3.13(c)). This ordering of units requires no long cables while still observing the physical constraint that units must be laid out in a line.

The folding and interleaving process may be extended to *N-dimensional* torus topologies by folding each axis in turn, as illustrated in Figure 3.14. Folding once along each axis eliminates long connections crossing from left to right, top to bottom and from the bottom-left corner to the top-right corner. Since all axes are orthogonal in non-hexagonal topologies, the folding process only moves units along the axis being folded. Unfortunately, this type of folding does not work for hexagonal torus topologies due to the non-orthogonality of the three axes. To exploit the folding technique used by non-hexagonal topologies, the units in a hexagonal torus topology must be mapped into a space with orthogonal coordinates. The choice of transformation to an orthogonal coordinate system can have an impact on how physically far apart logically neighbouring units are in the final arrangement.

Figure 3.15 illustrates the two transformations proposed by Heathcote [94] to map hexagonal arrangements of units into a 2D orthogonal coordinate space.

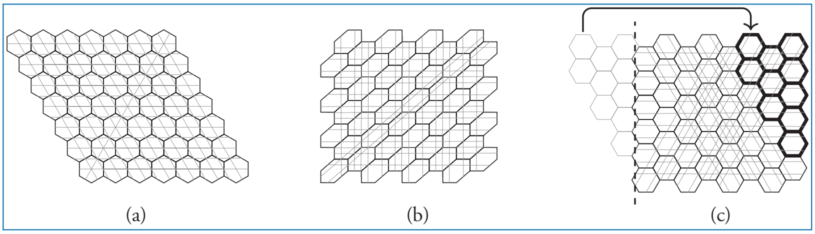


Figure 3.15. Transformations to map hexagonal arrangements of units into a 2D orthogonal coordinate space. Figures reproduced with permission from Heathcote [94].

The first transformation, *shearing* (Figure 3.15(b)), is general purpose but introduces some distortion. The second transformation, *slicing* (Figure 3.15(c)), is less general but can introduce less distortion than shearing and therefore may lead to shorter cable lengths.

Once a regular 2D grid of units has been formed, this may be folded in the conventional way as illustrated in Figure 3.14. Any shear-transformed network may be folded this way since its wrap-around connections always follow this pattern. Slice-transformed networks may only be folded like this when their aspect ratio is 1:2 when the pattern of wrap-around links is the same as a shear-transformed network. When 'square' networks, that is, those with a 1:1 aspect ratio, are sliced, the network must be folded twice along the Y axis to eliminate the criss-crossing wrap-around links. It is not possible to eliminate wrap-around links from sliced networks with other aspect ratios by folding. After folding, the units are interleaved, yielding a 2D arrangement of units in which no connection spans the width or height of the system. The maximum connection distance is constant for any network thus allowing the topology to scale up.

As indicated earlier, the hexagonal torus topology also applies to SpiNNaker when the boards are considered as nodes. The folded and interleaved arrangement of units produced by these techniques may be translated into physical arrangements of SpiNNaker boards in a machine room. Figure 3.16 illustrates how the SpiNN-5 boards that make up *SpiNNaker1M* can be folded and interleaved to keep cable length short.[1]

3.3.3 *SpiNNaker1M* Cabling

Due to the high density of units in a SpiNNaker system, the detailed cabling patterns used can be complex, despite their overall regularity. Figure 3.17 shows SpiNNaker team researchers and PhD students cabling *SpiNNaker1M*, a process that took several days even with the valuable cooperation of the machine itself.

[1] See also: https://youtu.be/z1_gE_ugEgE

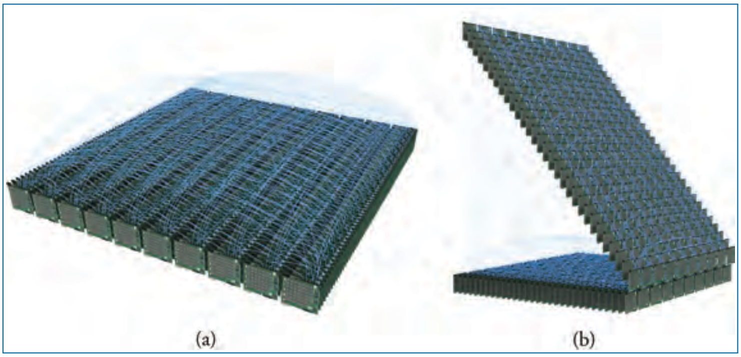


Figure 3.16. *SpiNNaker1M*: Long interconnect wires are avoided by folding and interleaving the board array in both dimensions.

Figure 3.17. *SpiNNaker1M* cooperates in its cabling (*l. to r.* Christian Brenninkmeijer, Robert James, Garibaldi Pineda García, Alan B. Stokes, Luca Peres and Andrew Gait). Luca earned the right to connect the last cable by providing the closest estimate to the total length of cable used (see Table 3.8).

To cope with this complexity, we developed SpiNNer [96], a collection of software tools for generating cabling plans and guiding cable installation and maintenance of SpiNNaker machines. SpiNNer employs diagnostic hardware built into SpiNNaker to guide the cable installation process. Figure 3.18(a) shows how SpiNNer uses diagnostic LEDs on the SpiNNaker boards to indicate where to connect a cable. The software also provides step-by-step cabling instructions via a Graphical User Interface (GUI), shown in Figure 3.18(b), and audible instructions delivered via headphones. These instructions explicitly specify the length of cable to use for each connection, thus avoiding the common problem of technicians over-estimating the cable length required [156]. Diagnostic registers in the

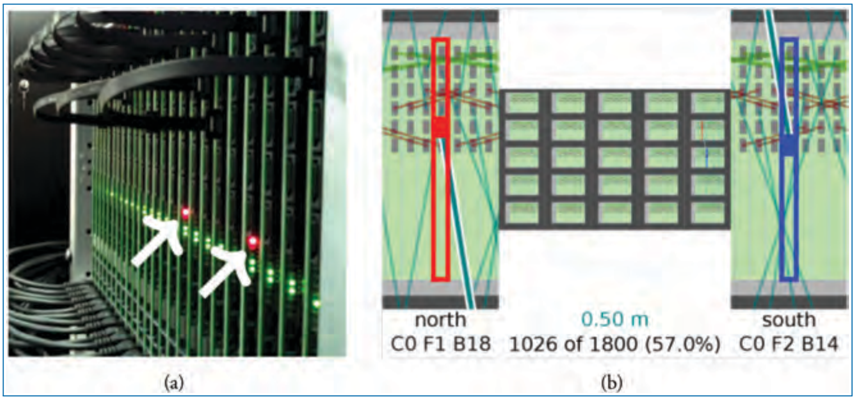


Figure 3.18. SpiNNer guides cable installation. Figures reproduced with permission from Heathcote [94].

Table 3.8. SATA cable usage in *SpiNNaker1M*.

Cable length (m)	Quantity	Total length (m)
0.15	304	45.60
0.30	1,504	451.20
0.50	1,014	507.00
0.70	742	519.40
0.90	36	32.40
total	3,600	1,555.60

spiNNlink interconnect are then used to verify the correct installation of each cable in real time, ensuring that mistakes are highlighted and fixed immediately.

The 'rule of (three-)thumbs' proposed by Mazaris [156] was used in *SpiNNaker1M*. This rule suggests that a minimum of 5 cm of cable slack should be provided. As SpiNNaker uses off-the-shelf SATA cables, only standard lengths were available. For any given span, the shortest length of cable providing at least 5 cm of slack was used. Table 3.8 lists the cable lengths used and the total number of cables of each length. The table shows a total cable length of over 1.5 km.

3.4 Using the Million-Core Machine: Tear it to Pieces

SpiNNaker1M is a massively parallel computer system that contains over one million energy-efficient ARM cores aimed at simulating one billion spiking neural

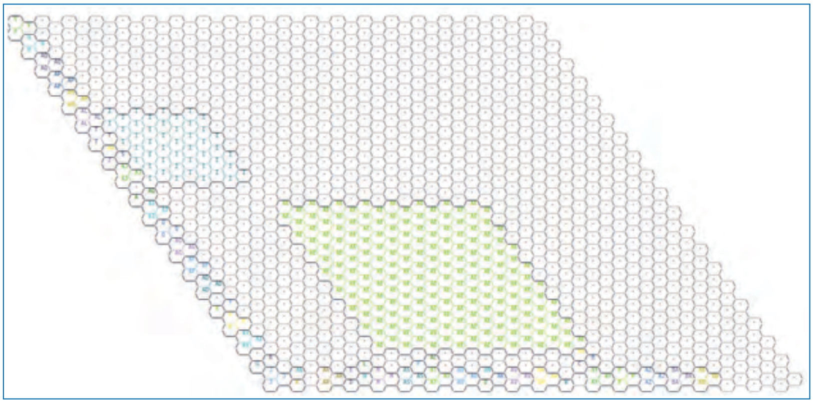


Figure 3.19. *SpiNNaker1M* partitioned into smaller *virtual machines*.

networks in biological real time. Most likely, though, not every simulation run on *SpiNNaker1M* will consist of a billion spiking neurons. To improve system throughput and energy efficiency, we developed a centralised software system which partitions large SpiNNaker machines into smaller ones on demand. This system is used to run many simulations in parallel on the same machine. The SpiNNaker machine partitioning and allocation server (Spalloc) [95] enables users to request *virtual* SpiNNaker machines of various shapes and sizes. These requests are queued and allocated in turn, partitioning *SpiNNaker1M* into the requested shape. Figure 3.19 shows a *SpiNNaker1M* diagram with various jobs allocated through Spalloc. Jobs can be as small as 1 SpiNN-5 board and as large as the whole machine, that is, 1,200 boards.

When faced with the numerous research problems of optimal packing and scheduling of allocations, this implementation uses the 'simplest mechanism that could possibly work'. This means that a job may end up with a larger machine than requested, to accommodate a selection of shape and size. Spalloc communicates with the BMPs of the allocated boards to disable the FPGAs in order to isolate the virtual machine from neighbouring boards that are not part of the machine. When a machine is allocated to a job it is powered on but not booted, that is up to the requester. This allows users complete control of the machine. The requester must keep the job alive by contacting the Spalloc server periodically and must release the allocated machine when finished.

3.5 *SpiNNaker1M* in Action

Figure 3.20 shows technician Dave Clark checking *SpiNNaker1M*, the 'million-core SpiNNaker machine'. The machine, located at the University of Manchester, was

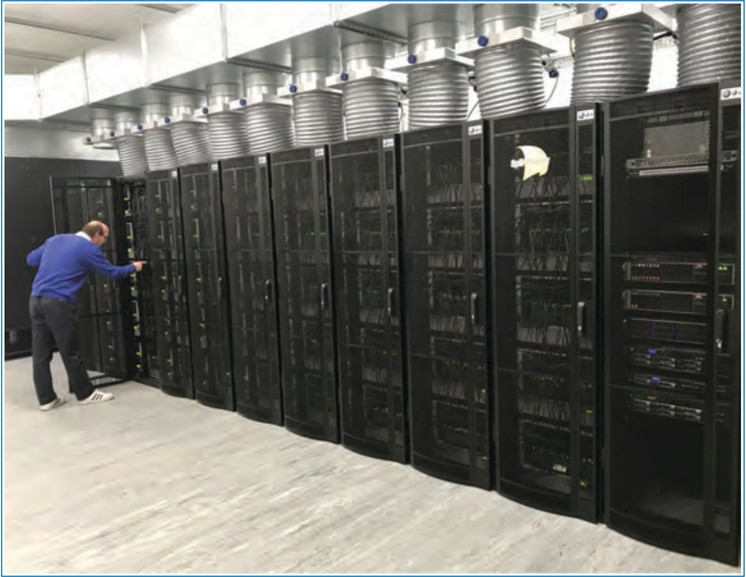


Figure 3.20. *SpiNNaker1M*: SpiNNaker million-core machine.

Figure 3.21. *SpiNNaker1M* inaugural boot.

commissioned on 2 November 2018 and serves researchers in the EU Human Brain Project (HBP). The cabinet at the right end of the photograph houses several host machines, including a server that provides remote access to *SpiNNaker1M* through the HBP portal and a second one that supports a Jenkins continuous integration testing platform for the SpiNNaker software tools described in Chapter 4.

Figure 3.21 shows a photograph of the celebration of the inaugural *SpiNNaker1M* boot on 2 November 2018, led by Prof. Steve Furber and Dr. Andrew Rowley. The machine reported 1,010,089 operational ARM cores, as shown in the photograph.

DOI: 10.1561/9781680836530.ch4

Chapter 4

Stacks of Software Stacks

By Andrew Rowley, Oliver Rhodes, Petruț Bogdan, Christian Brenninkmeijer, Simon Davidson, Donal Fellows, Steve Furber, Andrew Gait, Michael Hopkins, David Lester, Mantas Mikaitis, Luis Plana, and Alan Stokes

> *All hope abandon, ye who enter here.*
>
> — Dante Alighieri, Inferno

Alongside the job of designing and producing the hardware, there is the equally challenging task of constructing software that allows users to exploit the capabilities of the machine. Using a large parallel computing system such as SpiNNaker often requires expert knowledge to be able to create and debug code that is designed to be executed in a distributed and parallel fashion. More recently, software stacks have been created which try to abstract this process away from the end user by the use of explicit interfaces or by defining the problem in a form which is easier to map into a distributed system. In this chapter, we describe the SpiNNaker software stacks upon which most of the applications described in subsequent chapters are supported. It is built by merging slightly modified versions of the work presented by Rowley *et al.* [213], covering the software tools that allow the running of generic applications – the SpiNNaker Tools (SpiNNTools); and Rhodes *et al.* [207], covering the tools that specifically support the simulation of Spiking Neural Networks (SNNs) – the SpiNNaker backend for PyNN (sPyNNaker).

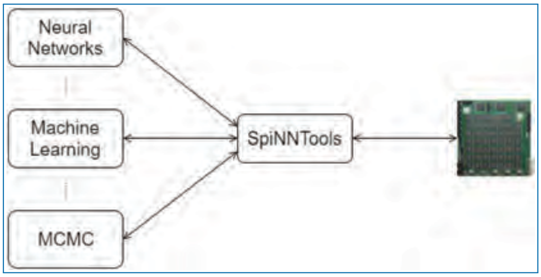


Figure 4.1. Applications using SpiNNTools to control SpiNNaker.

4.1 Introduction

A growing number of users are now using SpiNNaker for a wide range of tasks, including Computational Neuroscience [3] and Neurorobotics [1, 48, 209] for which the platform was originally designed, but also machine learning [240], and general parallel computation tasks, such as Markov Chain Monte Carlo inference computations [161]. The provision of a software stack for this platform aims to provide a base for the various applications, making it easier for them to exploit the full potential of the platform. Additionally, users will gain the advantage of any improvement in the underlying tools without requiring changes to their software (or at most only minor interface changes should they be required). A basic overview of this approach is seen in Figure 4.1.

The software stack allows the user to describe their computational requirements in the form of a graph, where the vertices represent the units of computation, and the edges represent the communication of data between the computational units. This graph is described in a high-level language and the software then maps this directly onto an available SpiNNaker machine. The SpiNNaker platform as a whole is intended to improve the overall execution time of the computational problems mapped onto it, and so the time taken to execute this mapping is critical; if it takes too long, it will dwarf the computational execution time of the problem itself.

The problem of writing code to run on the cores of the SpiNNaker machine is discussed in more detail by Brown *et al.* [25], along with the types of applications which might be suitable to execute on the platform. The software assumes that the application has already been designed to run in parallel on the platform; the SpiNNTools software then works to map that parallel application onto the machine, execute it and extract any results, along with any relevant data about the machine.

4.2 Making Use of the SpiNNaker Architecture

The nature of the SpiNNaker chip has important implications for the software running on the system. This section is a short recap of Chapters 2 and 3. Firstly, it must be possible to break up the computation of the application into units small enough that the code for each part fits on a single core. The SDRAM is shared between the cores on a single chip, and this property can be used by the application to allow cores to operate on the same data within the same chip. A small amount of data can be shared with cores operating on other chips as well through communication via the SpiNNaker router. The SpiNNaker boards can be connected together to form an even larger grid of chips, so appropriately parallelisable software could potentially be scaled to run on up to 1 million cores.

The SpiNNaker router is initially set up to handle the routeing of system-level data. The data to be sent by applications make use of the multicast packet type, meaning that a packet sent from a single source can be routed to multiple destinations simultaneously. To make multicast routeing work, the routeing tables of the router must be set up; this process is described in Section 4.7.

Each chip has an Ethernet controller, although in practice only one chip is connected to the Ethernet connector on each board. The chip with the Ethernet connected to it is then called the Ethernet chip, and this is used to communicate with the outside world, allowing, for example, the loading of data and applications. Communications with other chips on a board from outside of the machine must therefore go via the Ethernet chip; system-level packets are used to effect this communication between chips. In practice, the Ethernet connector of every board in a SpiNNaker machine is connected and configured, although this is not a requirement.

SpiNNaker machines are designed to be fault tolerant, so it is possible to have a functional machine with some missing parts. For example, it is normal that some of the SpiNNaker chips have 17 instead of 18 working cores, and sometimes even fewer than this as operational cores are tested more thoroughly than the testing done at manufacture. Additionally, machines can have whole chips that have been found to have faults, as well as some links broken between the chips and boards. The machine includes memory onto which faults can be stored statically in a *blacklist*, so that during the boot process these parts of the machine can be hidden to avoid using them.

SpiNNaker machines can be connected to external devices through either a SpiNNaker link connector, of which there is one on every 48-node board, or a spiNNlink SATA connector, of which there are 9 on each board; of those, 6 are used to connect to other boards. This, along with the low power requirements, makes the machine particularly useful for robotics applications, since the board can be connected directly to the robot without any need of other equipment. The only requirement is that the external devices must be configured to talk to the machine using SpiNNaker packets. The links can be configured to connect directly to a subset of the SpiNNaker chips on the board, and entries in the routeing tables of those chips can be used to send packets to any connected device and to route packets received from the devices across the SpiNNaker network.

4.3 SpiNNaker Core Software

The ARM968 cores can execute instructions from the ITCM using the ARM or Thumb instruction sets; generally, this code is generated from compiled C code using either the GNU's Not Unix (GNU) gcc compiler[1] or the ARM armcc compiler.[2] To this end, a library known as the SpiNNaker Application Runtime Kernel (SARK) has been written which allows access to the features of the SpiNNaker core and chip [25]. Additionally, software called the SpiNNaker Control And Monitor Program (SCAMP) has also been written which allows one of the cores to operate as a monitor processor through which the chip can be controlled [25], allowing, for example, the loading of compiled applications onto the other cores of the chip, the reading and writing of the SDRAM, the loading of the SpiNNaker routeing tables and, of course, controlling the operation of the chip's blinkenlight. SCAMP software can also map out parts of the machine known to be faulty when it is first loaded. Thus, when a description of the machine is obtained via SCAMP, only working parts should be present. The list of faults is stored on the boards themselves and can be updated dynamically if other parts are subsequently found to be faulty.

The SCAMP code can be loaded onto one core on every chip of the machine, and these cores then coordinate with each other allowing communication to any chip via any Ethernet connector on the machine (see below). This communication makes use of the SpiNNaker Datagram Protocol (SDP) [64], which is encapsulated into User Datagram Protocol (UDP) packets when going off machine to external devices. Communication out of the machine from any core is achieved by using Internet Protocol (IP) Tags. The SCAMP monitor processor on each Ethernet chip maintains a list of up to 8 IP Tags, which maps between values in the tag field of the

[1] https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

[2] https://developer.arm.com/products/software-development-tools/compilers/legacy-compiler-releases

SDP packets and an external IP address and port. When a packet is received that is destined to go out via the Ethernet (identified in the SDP packet header), this table is consulted and an UDP packet is formed containing the packet and this is sent to the IP address and port given in the table. The table can also contain Reverse IP Tags, where an UDP packet received from an external source is mapped from the UDP port in the packet to a specific chip and core on the machine, where the data of the packet are extracted and put into an SDP packet before being forwarded to the given core.

SARK provides a hardware abstraction layer, simplifying interaction with the DMA, network interface and communications controllers. SpiNNaker1 API (SpiN1API) provides an event-based operating system, as shown in Figure 4.16, with three processing threads per core: one for task queuing, one for task dispatch and one to service Fast Interrupt Request (FIQ). SpiN1API also provides the mechanism to link software callbacks to hardware events and enables triggering of actions such as sending a packet to another core and initiating a DMA. Callbacks are registered with different priority levels ranging from -1 to 2 depending on their desired function, with lower numbers scheduled preferentially. Callback tasks of priority 1 and 2 can be queued (in queues of maximum length 15), with new events added to the back of the queue. Callbacks of priority -1 and 0 are not queued, but instead pre-empt tasks assigned higher priority level numbers. Operation of this system follows the flow detailed in Figure 4.16(a).

The scheduler thread places callbacks in queues for priority levels 1 and above, and the dispatcher picks these callbacks and executes them based on priority. When the dispatcher is executing a callback of priority 1 or higher, and a callback of priority 0 is scheduled, this task pre-empts that currently being executed causing it to be suspended until the higher priority callback has completed. Callbacks of priority -1 use the FIQ thread to interact with the scheduler and dispatcher, enabling fast response and pre-emption of priority 0 and above tasks. Pointers are stored allowing fast access to the callback code, and the processor switches to FIQ banked registers to avoid the need for stacking [230], optimising the response time of priority -1 callbacks. However, this optimised performance limits the application to registering only a single -1 priority event and callback.

4.4 Booting a Million Core Machine

The process of booting the machine is shown in Figure 4.2. When the machine is first powered up, the cores on every chip start executing the boot ROM image. This is stored within the chip and cannot be altered. After testing the ITCM and

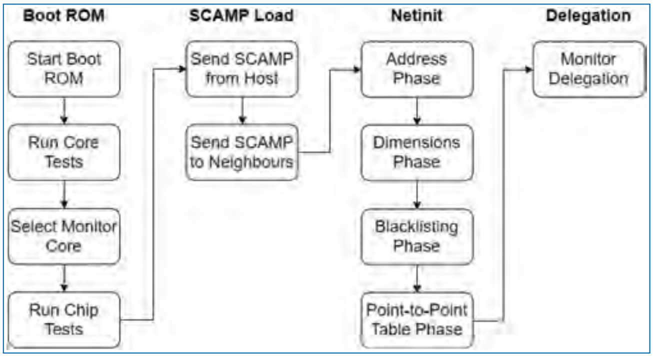


Figure 4.2. The stages of the SpiNNaker boot process.

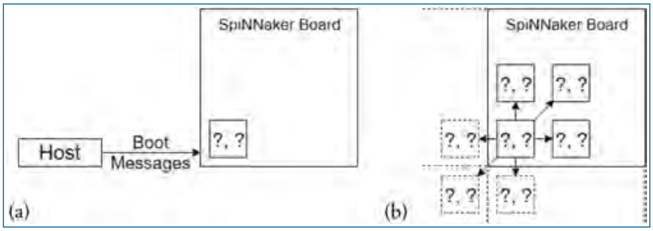


Figure 4.3. Booting SCAMP on the machine. (a) The SCAMP image is encoded in SpiNNaker boot messages and sent to the machine, where it is loaded on to the selected monitor processor of the Ethernet chip. (b) The SCAMP image is sent to neighbouring chips, which might include chips on adjacent boards, using NN packets.

DTCM of the core, the image then proceeds to determine if the core executing it is to be the monitor, through reading a mutex in the chip's *System Controller*; the first core to read this locks the mutex and so becomes the monitor. The processor selected as monitor now performs further tests on the shared parts of the chip.

Once the tests are complete, the Ethernet chips are set up to listen for boot messages being transmitted using UDP on port 54321. As shown in Figure 4.3(a), the host now sends the SCAMP image to one of these Ethernet chips; it is not critical which of these is selected, as the SCAMP software is set up to work out the dimensions of the machine and the coordinates once it has been loaded. The boot messages consist of a start command, followed by a series of 256-byte data blocks (with an appropriate header to indicate the order), followed by a completion command. If all the blocks are successfully received and assembled, the code stored in the data blocks is copied to the ITCM of the monitor processor and executed.

The current version of the SCAMP application starts with an initialisation phase where various parts of the hardware on the chip are set up for operation. The code is then transferred to all neighbouring chips using NN packets, as shown in Figure 4.3(b). Note that at this point, SCAMP does not know how many chips are there in the whole machine, and P2P routeing tables have not been initialised, so the only protocol available for communication between the chips is nearest neighbour. To this end, SCAMP establishes a protocol to determine whether to forward NN packets received to other neighbouring chips, and down which links.

Once the image has been transferred, the core now enters the 'netinit' stage, whereby communications with all other chips on the board is established, and the point-to-point routeing tables are built. This stage proceeds as follows:

1. *Address Phase*. During this phase, each SCAMP computes and sends out its computed coordinates based on the coordinates it receives from its neighbours; for example, if it receives [0, 0] from the 'west' link, it will assume that its coordinates are [0, 1], and if it receives [0, 0] from the 'north' link, it will assume its coordinates are [−1, 0] (coordinates are allowed to be negative at this stage). This phase continues until no new coordinates are received within a given time period.
2. *Dimensions Phase*. Each SCAMP sends its perceived dimensions of the machine based on the dimensions received from its neighbours. This again continues until no change of dimensions has occurred within a given time period.
3. *Blacklisting Phase*. The blacklist is sent from the Ethernet chip of each board to the other chips on the same board. This may result in the current monitor core discovering it is blacklisted. This is noted and delegation is then set up.
4. *Point-to-Point Table Phase*. Each SCAMP sends its coordinates once again, and these are forwarded on along with a hop count, so that every chip receives them eventually. These are used to update the point-to-point tables based on the direction in which the coordinates are received, along with the hop count to allow the use of the shortest route.
5. *Monitor Delegation Phase*. If the current SCAMP core has been blacklisted, it now delegates to another core that has not. This is done at this late stage to avoid interfering with the rest of the setup process.

Note that delegation of a blacklisted monitor core will not happen until after the 'netinit' phase has completed. The monitor core tends to be selected from a subset of the cores on the chip due to manufacturing properties; this means that boards where a core which is in this subset is so broken that it cannot perform the steps up to this point will not work with this system. A possible future change would therefore be to perform the blacklisting phase earlier in the process.

4.5 Previous Software Versions

Using SpiNNaker machines in the past required end users to load compiled applications and routeing tables manually onto the SpiNNaker machine through the use of the low level *ybug* software included with the aforementioned libraries.[3] Other software was then designed to ease the development of application code for end users. These consisted of: the aforementioned low-level libraries SARK and SpiN1API, and the monitor core software SCAMP, a collection of C code which represented models known in the neuroscience community and defined by the PyNN 0.6 language [44] and a collection of Python code which translates PyNN models onto a SpiNNaker machine. These pieces of software were amalgamated into a software package known as PACMAN 48 [68] and supported the main end-user community of computational neuroscientists for a number of years. These tools had the following limitations:

- They only supported SpiNNaker machines consisting of a single SpiNN-3 or SpiNN-5 board.
- They were designed to support only the computational neuroscience community, and thus, non-neural applications were not supported.
- End users were still expected to have expertise in using the SpiNNaker hardware. This was required as they were expected to run separate scripts manually, which together and in this order:
    1. Boot the SpiNNaker machine,
    2. Load executables onto the SpiNNaker machine,
    3. Load data objects onto SpiNNaker,
    4. Check when the executing code finished,
    5. Extract data from the SpiNNaker machine.

It was decided that a new software stack should be built to address these issues. The intention of this is to support a range of suitable applications executing on the SpiNNaker hardware by providing a flexible abstraction layer where the end user represents their problem as a graph, which is then executed on the SpiNNaker machine without requiring such a low-level knowledge of how the machine works, thus overcoming the issues mentioned above. This concept is briefly mentioned as ‘The Uploader’ by Brown *et al.* [25], although the framework is more complete in that it also:

- allows the user to express the generation of the data structures to be loaded (and possibly reloaded when changes have been made);

[3] Available from https://github.com/SpiNNakerManchester/spinnaker_tools/releases

- controls the execution flow of the application where required;
- aids in the storage and retrieval of data recorded during the execution;
- and extracts and presents provenance data which can be used to determine the correctness of the results.

4.6 Data Structures

4.6.1 SpiNNaker Machines

A SpiNNaker machine is represented as a set of Python classes as shown in Figure 4.4, with a main Machine class which then contains instances of classes for each of the parts of the machine represented. This data structure includes the important details of the machine for mapping purposes, including the chips, cores and links available, as well as the speed of each core, and the SDRAM available and the number of routeing entries available on each chip (in case some of this resource is used by the system software, as it is in the case of SCAMP). As well as internally representing a physical, real-world machine with all its faults mapped out, this representation also allows the instantiation of a virtual machine for testing in the absence of connected hardware. The virtual machine can be further modified to simulate hardware faults and analyse software behaviour.

The connection of external devices, such as a silicon retina or a motor to the machine, is represented using 'virtual chips'. A virtual chip will be given coordinates of a chip that does not exist in the physical machine and is therefore marked as virtual. The coordinates do not have to align with the rest of the machine, as the location where the chip is connected to the other real chips in the machine is also identified. This allows any algorithm to detect that virtual chips are present if

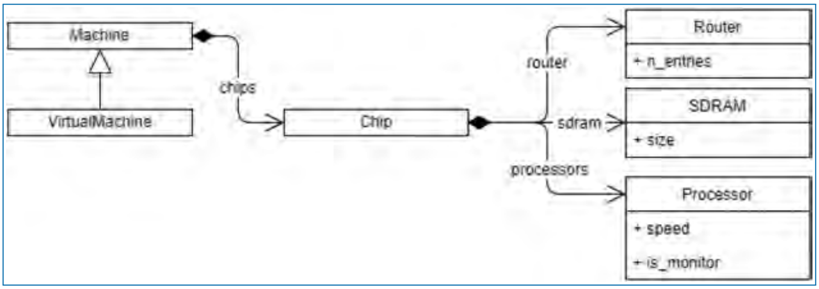


Figure 4.4. The Python class hierarchy for SpiNNaker Machine representation. The machine contains a list of chips, and each chip contains a router, an SDRAM and a list of processor objects, each with their respective properties. A *VirtualMachine* can also be made, which contains the same objects but can be identified as being virtual by the rest of the tools.

necessary and also to know where the connected real chip is to make use of that if needed.

4.6.2 Graphs

A graph in SpiNNTools consists of vertices and directed edges between the vertices. The vertex is considered to be a place where computation takes place, and as such, each vertex has a SpiNNaker executable binary associated with it. An edge represents some communication that will take place from a source, or pre-vertex to a target, or post-vertex. An additional concept is that of the outgoing edge partition; this is a group, or partition, of edges that all start at the same pre-vertex, as shown in Figure 4.5(b). This is useful to represent a multicast communication. Note that not all edges that have the same pre-vertex have to be in the same outgoing edge partition; there can be more than one outgoing edge partition for each source vertex. This represents different message types, which might be multicast to different sets of target vertices. Thus, each outgoing edge partition has an identifier, which can be used to identify the type of message to be multicast using that partition.

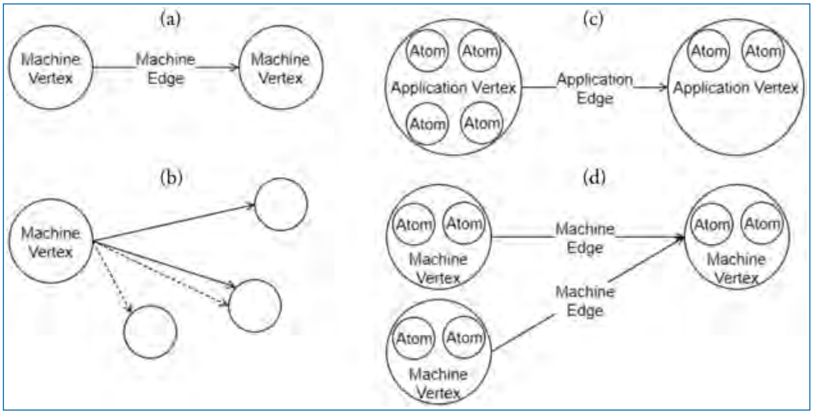


Figure 4.5. Graphs in SpiNNTools. (a) A Machine Graph made up of two Machine Vertices connected by a Machine Edge, indicating a flow of data from the first to the second. (b) A Machine Vertex sends two different types of data to two subsets of destination vertices using two different Outgoing Edge Partitions, identified by solid and dashed lines respectively. (c) An Application Graph made up of two Application Vertices, each of which contain two and four atoms, respectively, connected by an Application Edge, indicating a flow of data from the first to the second. (d) A Machine Graph created from the Application Graph in (c) by splitting the first Application Vertex into two Machine Vertices which contain two atoms each. The second Application Vertex has not been split. Machine Edges have been added so that the flow of data between the vertices in still correct.

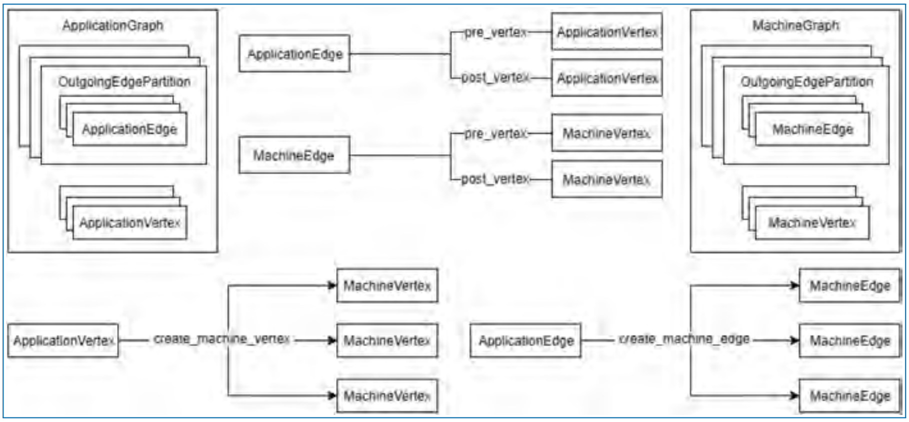


Figure 4.6. The relationship between the graph objects. An ApplicationGraph contains ApplicationVertex objects and OutgoingEdgePartition objects, which contain ApplicationEdge objects in turn. A MachineGraph similarly contains MachineVertex objects and OutgoingEdgePartition objects, which contain MachineEdge objects in turn. ApplicationEdge objects have pre- and post-vertex properties which are ApplicationVertex objects, and similarly MachineEdge objects and pre- and post-vertex properties which are MachineVertex objects. An ApplicationVertex can create a number of MachineVertex objects for a subset of the atoms contained therein and an ApplicationEdge can create a number of MachineEdge for a subset of atoms in the pre- and post-vertices.

There are two types of graph represented as Python classes in the tools (a diagram can be seen in Figure 4.6). A Machine Graph, an example of which is shown in Figure 4.5(a), is one in which each vertex (known as a Machine Vertex) is guaranteed to be able to execute on a single SpiNNaker processor. A Machine Edge therefore represents communication between cores. In contrast, an Application Graph, an example of which is shown in Figure 4.5(c), is one where each vertex (known as an Application Vertex) contains atoms, where each atom represents an atomic unit of computation into which the application can be split; it may be possible to run multiple atoms of an Application Vertex on each core. Each edge (known as an Application Edge) represents communication of data between the groups of computational units; if one or more of the atoms in an Application Vertex communicates with one or more atoms in another Application Vertex, there must be an Application Edge between those Application Vertices. It is not guaranteed that all the atoms on an Application Vertex fit on a single core, so the instruction code for Application Vertices should know how to process a subset of the atoms, and how to handle a received message and direct it to the appropriate atom or atoms. The graph classes support adding and discovering vertices, edges and outgoing edge partitions.

As the vertices represent the application code that will run on a core, they have methods to communicate their resource requirements, in terms of the amount of DTCM and SDRAM required by the application, the number of Central Processing Unit (CPU) cycles used by the instructions of the application code to maintain any time constraints, and any IP Tags or Reverse IP Tags required by the application. The Application Vertex provides a method that returns the resources required by a continuous range or slice of the atoms in the vertex; this is specific to the exact range of atoms, allowing different atoms of the vertex to require different resources. The Application Vertex additionally defines the maximum number of atoms that the application code can execute at a maximum on each core of the machine (which might be unlimited) and also the total number of atoms that the vertex represents. These allow the Application Vertex to be broken down into one or more Machine Vertices as seen in Figure 4.5(d); to this end, the Application Vertex class has a method for creating Machine Vertex objects for a continuous range of atoms. A Machine Vertex can return the resources it requires in their entirety.

The graphs additionally support the concept of a Virtual Vertex. This is a vertex that represents a device connected to a SpiNNaker machine. The Virtual Vertex indicates which chip the device is physically connected to, allowing the tool chain to work with this to include the device in the network. As with the other vertices, there is a version of the Virtual Vertex for each of the machine and application graphs.

4.7 The SpiNNTools Tool Chain

The aim of the SpiNNTools tool chain is to control the execution of a program described as a graph on the SpiNNaker machine. The software is executed in several steps as shown in Figure 4.7 and detailed below.

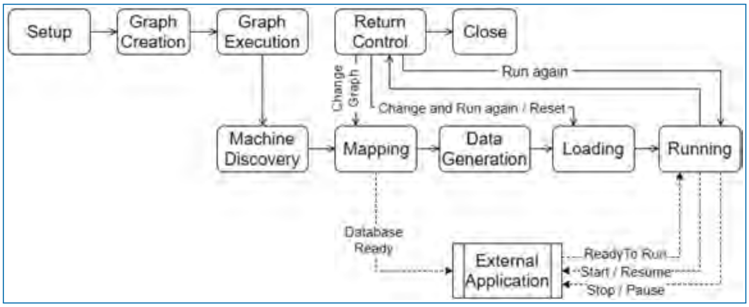


Figure 4.7. The execution work flow of SpiNNTools in use within an application. Once control has returned to the application, the flow can be resumed at different stages depending on what has changed since the last execution.

4.7.1 Setup

The first step in using SpiNNTools is to initialise them. At this point, the user can specify appropriate configuration parameters, such as the time step of the simulation, and the location where binary files can be located on the host machine. The tool chain then sets up the initially empty graphs and reads in configuration files for further options, such as the SpiNNaker machine to be used. Options are separated out in this way to allow script-level parameters which might apply no matter where the script is run (like the timestep of the simulation), from user-level parameters, which will be different per-user, but likely to be common across multiple scripts for that user (like the SpiNNaker machine to be used).

4.7.2 Graph Creation

Once the tool chain has been initialised, the user can add vertices and edges to either an application or machine graph. It is an error to add vertices or edges to both of these structures. The tool chain keeps track of the graph as it is built up. Users can extend the vertex and edge classes to add additional information relevant to their own application.

4.7.3 Graph Execution

Once the user has built their graph, they then call one of the methods provided to start execution of the graph. Methods are provided to run for a specified period of time, to run until a completion state is detected (such as all cores being in an exit state having completed some unit of work), or to run 'forever' meaning that execution can be stopped through a separate call to SpiNNTools at some indeterminate time in the future, or the execution can be left on the machine to be stopped outside of the tool chain by resetting the machine. The graph execution itself consists of several phases shown in the lower half of Figure 4.7 and detailed below.

Machine Discovery

The first phase of execution is the discovery of the machine to be executed on. If the user has configured the tool chain to run on a single physical machine, this machine is contacted, and if necessary booted. Communications with the machine then take place to discover the chips, cores and links available. This builds up a Python machine representation to be used in the rest of the tool chain.

If a machine is to be allocated, SpiNNTools must first work out how big a machine to request, by working out how many chips the user-specified graph requires. If a machine graph has been provided, this can be used directly, since the number of cores is exactly the number of vertices in the graph. The resources must still be queried, as the SDRAM requirements of the vertices might mean that not all of the cores on each chip can be used. For example, a graph consisting of 10 machine vertices, each requiring 20 MByte of SDRAM and thus 200 MByte of SDRAM overall, will not fit on a single chip in spite of there being enough cores.

If an application graph is provided, this must first be converted into a machine graph to determine the size of the machine. This is done by executing some of the algorithms in the mapping phase (see below).

Mapping

The mapping phase takes the graph and maps it onto the discovered machine. This means that the vertices of the graph are assigned to cores on the machine, and edges of the graph are converted into communication paths through the machine. Additionally, other resources required by the vertices are mapped onto machine resources to be used within the simulation.

If the graph is an application graph, it must first be converted to a machine graph. This may have been done during the machine discovery phase as described previously. To allow this, the algorithm(s) used in this 'graph partitioning' process are kept separate from the rest of the mapping algorithms.

Once a machine graph is available, this is mapped to the machine through a series of phases. This must generate several data structures to be used later in the process. These include:

- a set of *placements* detailing which vertex is to be run on which core of the machine;
- a set of *routeing tables* detailing how communications over edges are to pass between the chips of the machine;
- a set of *routeing keys* detailing the range of keys that must be sent by each vertex to communicate over each outgoing edge partition starting at that vertex;
- a set of *IP tags* and *reverse IP tags* identifies which external communications are to take place through which Ethernet-connected chip.

Note that once machine has been discovered, mapping can be performed entirely separately from the machine using the Python machine data structures created. However, mapping could also make use of the machine itself by executing specially designed parallel mapping executables on the machine to speed up the execution. The design of these executables is left as future work.

Mapping information can be stored in a database by the system. This allows for external applications which interact with the running simulation to decode any live data received. As shown in Figure 4.7, the applications can register to be notified when the database is ready for reading and can then notify SpiNNTools when they have completed any setup and are ready for the simulation to start, and when the simulation has finished.

Data Generation

The data generation phase creates a block of data to be loaded into the SDRAM for each vertex. This can be used to pass parameters from the Python-described vertices to the application code to be executed on the machine. This can make use of the mapping information above as appropriate; for example, the *routeing keys* and *IP tags* allocated to the vertex can be passed to ensure that the correct keys and tags are used in transmission. The graph itself could also be used to determine which *routeing keys* are to be received by the vertex, and so set up appropriate actions to take upon receipt of these keys.

Some support for data generation and reading is provided by the tool chain both at the Python level, where data can be generated in 'regions', and at the C code level, where library functions are provided to access these regions. Other more basic data generation is also supported which simply writes to the SDRAM directly.

Data generation can also create a statistical description of the data to be loaded and then expand these data through the execution of a binary on the machine. This allows less data to be created at this point potentially speeding up the data generation and loading processes, and also allows the expansion itself to occur in parallel on the machine.

Loading

The loading phase takes all the mapping information and data generated, along with the application binaries associated with each machine vertex, and prepares the physical machine for execution. This includes loading the *routeing tables* generated on to each chip of the machine, loading the application data into the SDRAM of the machine, loading the *IP tags* and *reverse IP tags* into the Ethernet chips, and loading the application code to be executed.

Running

The running phase starts off the actual execution of the simulation and, if necessary, monitors the execution until complete. Before execution, the tool chain wait for the completion of the setup of any external applications that have registered to read the mapping database. These tools are then notified that the application is about to start, and when it is finished.

Once a run is complete, application recorded data and provenance data are extracted from the machine. The provenance data include:

- router statistics, including dropped multicast packets;
- core-level execution statistics, including information on whether the core has kept up with timing requirements;

- custom core-level statistics, these depend on the application, but might include such things as the number of spikes sent in a neural simulation or the number of times a certain condition has occurred.

The log files from each core can also optionally be extracted. During provenance extraction, each vertex can analyse the data and report any anomalies. If the log files have been extracted, these can also be analysed and any 'error' or 'warning' lines can then be printed.

If a run is detected to have failed in some way, the tool chain will attempt to extract information about this failure. A failure includes one of the cores going into an error state, or if the tool chain have been run for a specific duration, if the cores are not in a completion state after this time has passed. Log files will be automatically extracted here and analysed as previously discussed. Any cores that are still alive will also be asked to stop and extract any provenance data so that this can also be analysed in an attempt to diagnose the cause of the error.

The run may be split into several sub-runs to allow for the limited SDRAM on the machine, as shown in Figure 4.8. After each run cycle, any recorded data are extracted from the SDRAM and stored on the host machine, after which the recording space is flushed, and the run cycle restarted. This requires additional support within the binary of the vertex, to allow a message to be sent to the core to increase the run duration, and to reset the recording state. This support is provided in the form of C code library functions, with callbacks to allow the user to perform

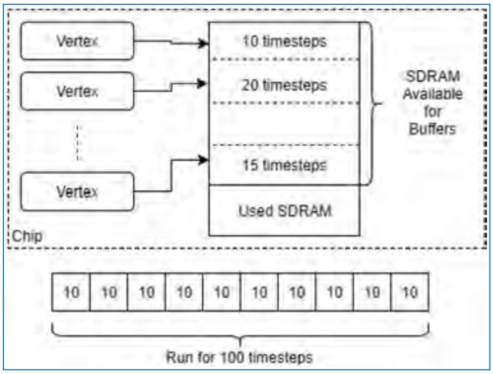


Figure 4.8. Running vertices with recorded data. The SDRAM remaining on each chip after it has been allocated for other things is divided up between the vertices on that chip. Each vertex is then checked for the number of time steps it can be run for before filling up the SDRAM. The minimum number of time steps is taken over all chips and the total run time is split into smaller chunks, between which the recorded data are extracted and the buffer is cleared.

additional tasks before resuming execution at each phase. Additionally, the tool chain can be set up to extract and clear the core logs after each run cycle to ensure that the logs do not overflow.

The length of each run cycle can be determined automatically by SpiNNTools. This is done by working out the SDRAM available on each chip after data generation has taken place. This free space is then divided between the vertices on the chip depending on how much space they require to record per time step of simulation. To ensure that there is some space for recording, the user can specify the minimum number of time steps to be recorded and space for this is allocated statically during the mapping phase (noting that if this space cannot be allocated, this phase will fail with an error).

At the end of each run phase, external applications are notified that the simulation has been paused and are then notified again when the simulation resumes. This allows them to keep in synchronisation with the rest of the application.

4.7.4 Return of Control/Extraction of Results

Once the run cycles have completed, the tool chain returns control to the executing script. At this point, the user can interact with the graph again. This includes the ability to extract any recorded data (see later) or make changes to the graph and/or the parameters before resuming the simulation. The effect of any changes is detailed below.

4.7.5 Resuming/Running Again

The user can choose to resume the execution of the simulation or to reset the simulation and start it again. At this point, the tool chain must decide which of the aforementioned steps need to be run again. If no changes have been made to the graph or the parameters, this can simply be considered an extension of the aforementioned ability of the SpiNNTools to run the code in phases. The minimum time calculated previously is respected again here and the tool chain will then run in cycles of this unit of time. Note that this means that if the first run-time is shorter than that required to fill the remaining SDRAM space (and thus only one run cycle was required previously), this time is taken as the minimum. This is because the buffers will have already been initialised to record for this amount of time. An extension to this work then is to allow the buffers to be sized to use up all of the remaining SDRAM regardless of the run time and then allow runs in units of less than or equal to the time that uses all of this space.

If the parameters of any of the vertices or edges have been changed, the vertex can be set up to allow the reloading of these changes. It is expected that this can be supported where the change will not increase the size of the data, and so can overwrite the existing data, such as a change in neuron state update parameters in a neural network. Any increase in the size of the data, such as an increase in the number of synapses in a neural network, would likely require a remapping of the graph on to the machine as the SDRAM is likely to be packed in such a way as to not allow the expansion of the data for a single core; it is left to the vertex to make this decision however.

Any change to the graph, such as the addition of a vertex or edge, is likely to require that the mapping phase take place again. This may even result in a new machine being required should the size of the graph increase to this degree. This will mean that all the other phases will also have to be executed again.

4.7.6 Closing

Once the user has finished simulating and extracted any data, they can tell the tool chain that they are finished with the machine by closing it. At this point, the tool chain resets and releases any machines that have been reserved, and so recorded data will no longer be available. If the tool chain was told to run the network for an indeterminate length, this would also result in the extraction and evaluation of any provenance data at this stage.

4.7.7 Algorithms and Execution

To run each of the above phases, SpiNNTools executes a series of algorithms. The algorithms consume various inputs that are made available by the tool chain and by other algorithms, and produce various outputs. These inputs and outputs are not constrained in any other way; thus, algorithms are not constrained to produce only one output. This could be useful in, for example, mapping, where an algorithm could be made to produce both placements and routeing tables which have been optimised together. This is in contrast to restricting the algorithms to specific tasks, where the output might then be less optimal, such as having a specific algorithm for generating placement and another for generating routeing tables.

To support this form of execution, SpiNNTools implements a workflow execution system, shown in Figure 4.9. This examines the algorithms to be run in terms of the inputs required and outputs generated to compute an execution order for the algorithms. Input and output 'tokens' are also supported; these indicate implicit inputs and outputs; for example, a token might be used to represent that data have been loaded on to the machine, and thus, an algorithm can generate this as an output, and another can require that this has been completed before execution.

The algorithms themselves are not discussed here in detail other than those mentioned above. A more detailed discussion of the mapping algorithms is discussed

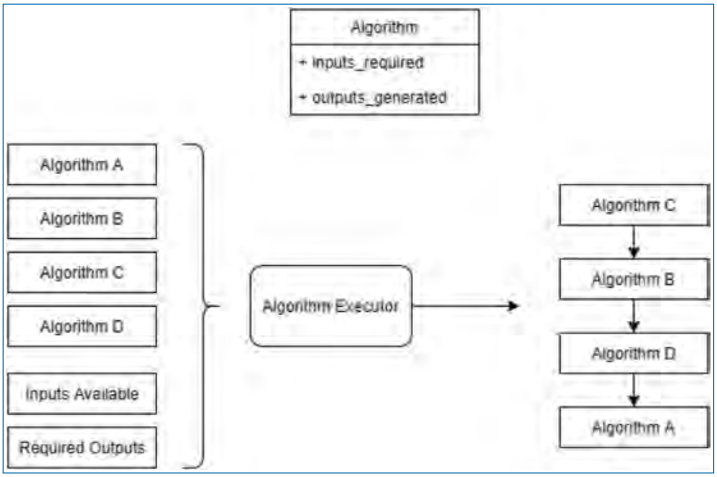


Figure 4.9. Algorithms being run by the algorithm execution engine. The executor is provided with a list of algorithms to run, a set of input items and a set of output items to produce. It then produces a workflow for the algorithms accounting for their inputs required and outputs produced.

by Heathcote [94]. The tool chain also includes algorithms for routeing table compression, which are discussed by Mundy *et al.* [174]. Many of the other algorithms are currently simplistic in nature; these can be replaced in the future should other algorithms be found to perform more efficiently and/or effectively.

4.7.8 Data Recording and Extraction

As mentioned previously, the tool chain supports the recording of data in such a way as to cope with the limited nature of the SDRAM on the machine. A ‘buffer manager’ is provided, which is used to keep track of and store the buffers of data as they are extracted from the machine. This can additionally support the live extraction of buffers whilst the simulation is running, as shown in Figure 4.10 (Top); cores configured with the provided library can contact the host machine when the recording space is getting full and the tool chain can then attempt to extract the data. In general, the bandwidth of the Ethernet of the machine is not fast enough for this to be effective, and data tend to be lost.

The SCAMP software supports the reading of SDRAM through SDP messages. This works through a request and response system, where each SDP message can request the reading of up to 256 bytes of data. Additionally, to transmit the SDP message to chips which are not connected to the Ethernet, this message must be broken down into SpiNNaker network messages and then reconstructed on receipt; an overview of how this process works is shown in Figure 4.10 (Middle). This

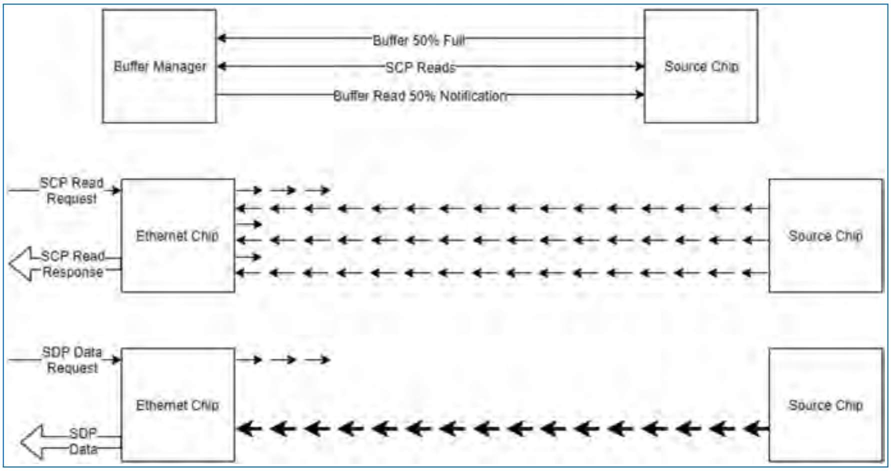


Figure 4.10. Data buffering and extraction. *Top:* The buffer manager is used to read back recorded data during execution; when the buffer contains some data, the buffer manager is notified and attempts to read the data, notifying the data source once this has been done to allow the space to be reused. *Middle:* Data reading done using SCAMP; each read of up to 256 bytes is further broken down into a number of request and read cycles on the machine itself, where the packets used contain only 24 bits of data each. *Bottom:* Data reading done using multicast messages; the initial request is all that is required, after which the data are streamed using packets containing 64 bits of data. The machine is set up so that these packets are guaranteed to arrive, so no confirmation is required.

results in speeds of around 8 Mb/s when reading from the Ethernet chip and around 2 Mb/s when reading from other chips.

To speed up the extraction of data, the tool chain includes the ability to circumvent this process, an overview of which is shown in Figure 4.10 (Bottom). To facilitate this, firstly the machine is configured so that packets can be sent with a guarantee that none of them are ever dropped; this can be done in this scenario because exactly one path through the machine will be used by each read, so deadlocks cannot occur. Next, one of the cores on each chip is loaded with an application that can read from SDRAM and stream multicast messages to another application loaded onto a core on the Ethernet chip, which then forms these into SDP messages to be streamed to the host along with a sequence number in each SDP packet. The host then gathers the SDP packets and notes which sequences are missing. The missing sequences are then requested again from the machine; this is repeated until all sequences have been received. This has numerous advantages over the SDP request-and-response mechanism: the SDP is only formed at the Ethernet chip, and thus, the headers do not get transmitted across the SpiNNaker fabric; and the host only sends in a single request for data and then a single request for each group of missing sequences and thus does not have to wait for each chunk of 256 bytes between sending requests. This results in speeds of up to 40 Mb/s when reading from any chip on the machine; there is no penalty for reading from a non-Ethernet chip.

Once this protocol was implemented, we discovered that the Python code had trouble keeping up with the speed at which the data were received from the machine. We therefore implemented a version of the data reception in C++ and Java that could interface with the Python code; the Java version is the version used in production following comparative testing and assessment of the integration quality. This then allows the use of the Ethernet connection on multiple boards simultaneously, allowing the data extraction speed to scale with the number of boards required for the simulation, up to the bandwidth of the network connected to the host machine.

4.7.9 Live Interaction

We have previously mentioned that external applications can interact with a live simulation, making use of the mapping database. Additional support for this interaction is provided by the tool chain. This support is split into live data output and live data input.

Live data output support is performed by a vertex called the 'Live Packet Gatherer', which will package up any multicast packets it receives and send them as UDP packets using the EIEIO protocol [205]. It is configured by adding edges to the graph from vertices that wish to output their data in this way. This has the advantage of being able to tap into the existing multicast streams that are already being used to communicate within the machine; this same data can be sent out of the machine by the simple addition of an edge to the graph, as shown in Figure 4.11.

Live data input support is provided via a vertex called the 'Reverse IP Tag Multicast Source', which will unpack and send multicast packets using the same EIEIO protocol. As with the Live Packet Gatherer, this vertex can then be configured by simply adding edges from it to the vertices which are to receive the messages.

External applications that would like to make use of this support can read the mapping database to determine the multicast keys to be received in the case of live output or to be sent in the case of live input. Support for this interaction is provided in SpiNNTools in both Python code and host-based C++ code.

4.7.10 Dropped Packet Re-Injection

As mentioned in Section 4.2, when a packet is dropped, an interrupt is raised allowing a core to detect and capture the dropped packet. The tool chain includes software that runs on the SpiNNaker machine to detect this interrupt and then

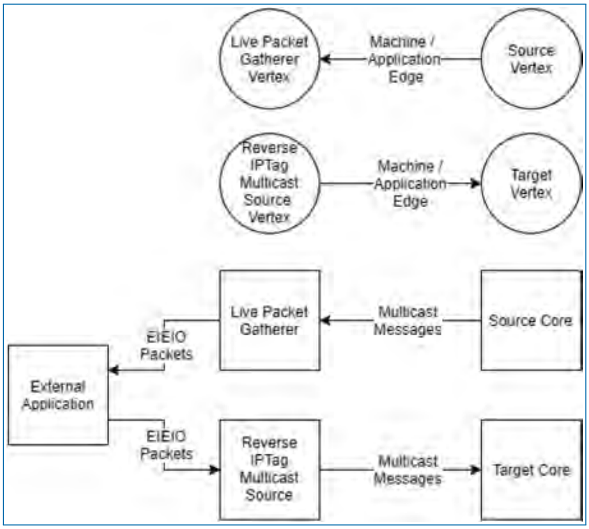


Figure 4.11. Live interaction with vertices. *Top:* To indicate that live output is required, an edge is added from the vertex which is the source of the data to the Live Packet Gatherer vertex in the graph. To indicate that the live input is required, an edge is added from the Reverse IP Tag Multicast Source vertex to the target of the data in the graph. *Bottom:* The effect of adding the edges to the graph is that multicast messages will be sent from the core (or cores) of the source vertex to the core running the Live Packet Gatherer, which will then wrap the messages in EIEIO packets and forward them to a listening external application; and EIEIO packets received from an external application will be decoded by the Reverse IP Tag Multicast Source core and dispatched as multicast messages to the target core (or cores).

capture the packets that have been dropped. These are stored until a time at which the router is no longer blocked and so can safely send the packet onwards. This helps in those applications where the reliable transmission of packets is critical to their operation.

There is only one register within the SpiNNaker hardware to hold a dropped packet. If a second packet is dropped, this packet will be completely unrecoverable; an additional flag is set in this scenario so the re-injection core can detect this and count such occurrences. This count is reported to the user at the end of the execution so that they know that something may not be correct in their simulation results.

4.7.11 Network Traffic Visualisation

A real-time traffic visualiser for a single 48-node SpiNN-5 board was developed to explore the control and monitoring of the SpiNNaker system in real time [144].

The visualiser shows the system traffic status by gathering and displaying data from the monitoring and profiling counters on the SpiNNaker chips in the system. The visualiser can also send commands to the monitor processor via the Ethernet connection to control and interact with the system.

4.7.12 Performance and Power Measurements

The tool chain includes support for profiling any executed code and for making an estimate of the power usage of a simulation. Profiling support is provided through both C and Python libraries, where the former is used to instrument code with 'entry' and 'exit' markers for code to be timed, and the latter is used to extract the recorded timing data and calculate various statistics on the run.

To provide a reasonably accurate power estimation, the tool chain includes support for sampling the *System Controller* to determine whether each core is busy or idle (waiting for an event to occur), and we include a uniformly-distributed random delay to the sampling to avoid the worst effects of sample aliasing. As this is run on the machine, it can achieve a higher sampling rate than a commercial power-measurement tools. We then use the proportion of time spent idling, together with the number of SpiNNaker messages sent, to compute the estimate for how much power was actually used, scaling by the previously measured long-term average power consumption per core and per message. This has been tested against a commercial power measurement device on a 24-board system and appears to provide results close to the real numbers.

4.8 Non-Neural Use Case: Conway's Game of Life

Conway's Game of Life [71] consists of a collection of cells which are either alive or dead based on the state of their neighbouring cells. A diagram of an example Machine Graph of this problem is shown in Figure 4.12. The vertices of the graph of this application are each a cell in the game; given the state of the surrounding cells, this cell can compute whether it is dead or alive in each step and then send that to its neighbours. It similarly receives the state of the neighbours as they are transmitted and again uses this to update its own state. The edges of the graph are thus between adjacent cells in a grid, where each vertex is connected bidirectionally to its eight surrounding neighbours. The game proceeds in synchronous phases, where the state of cells in a given phase are all considered at the same time.

Graphs of this form are highly scalable on the SpiNNaker system, since the computation to be performed at each node is fixed, and the communication forms a regular pattern which does not increase as the size of the board grows. Thus once

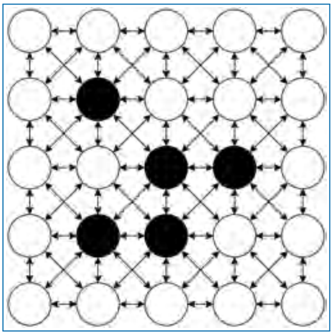


Figure 4.12. Conway's Game of Life on a 5 × 5 grid as a Machine Graph. Every Machine Vertex is connected to each of it's 8 neighbours bi-directionally; this requires two Machine Edges for each bi-directional connection. The initial state of each Vertex is either alive (black) or dead (white).

working, it is likely that any size of game can be built, up to the size of the available machine. This type of graph would also likely be suited to finite element analysis [17] problems, provided that the data to be transmitted can be broken down into SpiNNaker packets. This problem thus works well as an archetype.

It will be assumed that we have built the application code which will update the cell based on the state of the surrounding cells. This will update the state once per time step of the simulation based on the received state from the surrounding cells and then send its own new state out using the given key. It can also record its state at each time step in the simulation. The set-up of this application is as follows:

- A Conway vertex is created which extends the machine vertex class.
- A number of Conway vertices are added to the graph to make up the board. These are stored in such a way that finding an adjacent vertex in the grid is easy.
- A machine edge is added between each pair of adjacent vertices, in each direction.
- Each machine vertex generates data for the vertex, which includes the key to be sent by that vertex and the number of time steps to run for.
- Each machine vertex can tell the tool chain how many time steps it can run for given an amount of SDRAM available for recording.
- Each machine vertex contains code to read the state that is recorded at each time step using the Buffer Manager.

Once the graph is built, the script starts the execution of the graph. During this execution, the tool chain will obtain a machine description and use this with the machine graph to work out a placement of each of the vertices and a routeing of the edges between these placements, along with an allocated key for each of the vertices. The software tools will then ask each vertex how many time steps it can record for based on the available SDRAM after placement is complete, and the resources used on each chip can therefore be determined. Each vertex will then be asked to generate its data based on the mapping and timing information. SpiNNTools will then load the generated data onto the machine along with the routeing tables and application code and start the execution of the cores. It will wait an appropriate amount of time for the cores to stop and then check their status. Assuming this is successful, control will return to the script. This can then request the recorded states from each of the vertices and display these data in an appropriate way.

A future version could have a Conway vertex that can have multiple cells within each machine vertex, which would then allow for an application vertex of cells. This would have a single large Application Vertex which would represent the whole game board and an Application Edge for each of the 8 directions of connectivity, each in its own Outgoing Edge Partition to indicate that different keys are required for each of the directions. This would require that the vertex would have to cope with the reception of multiple neighbour states, which would make the application code itself more complex; for example, it would have to cope with multiple incoming keys from each direction, each of which would target a different cell within the grid.

Another possible extension to this application is to extract the state during execution and display this as the application progresses. This would require the addition of the Live Packet Gatherer vertex (described above) to the graph and an edge from each of the Conway vertices to this vertex. The script would then indicate, before executing the graph, that there is an external application that would like to receive the data. This application will receive a message when the mapping database has been written, at which point, it can set up a mapping between multicast keys received and positions in the game board, responding when it has completed its own setup. The tool chain will then notify this application that the simulation is starting, and the application will then receive the same state messages as the vertices receive, which it can use to update the display of the game board.

4.9 sPyNNaker – Software for Modelling Spiking Neural Networks

The SpiNNaker machine is primarily designed to simulate spiking neural networks [65]. As an example, we consider the simulation of a cortical column found within mammalian brains, that is, a model of the neurons within a structure underneath

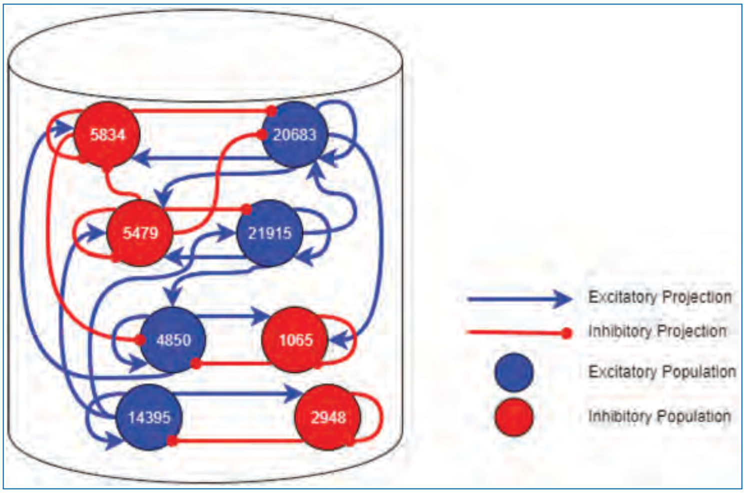


Figure 4.13. A neural network topology of a 1 mm^2 area of cortical microcircuit found within the mammalian brain. Each population of neurons is shown as a circle containing a number, where the number indicates the number of neurons in that population.

a 1 mm^2 area of the surface of the generic early sensory cortex [201]. Figure 4.13 shows the groups of neurons (Populations) in this network and the connectivity between them (Projections). In a spiking neural network, the vertices are groups of point neurons (as a single core can simulate more than one neuron); the computation required is the update of the neuron state in response to spikes received from connected neurons. The edges are then groups of synapses between the neurons, over which spikes are transmitted. These are potentially unidirectional and are likely to be more heterogeneous in nature than the regular grid pattern seen in Conway's Game of Life.

The problem of SNNs is clearly well suited to the architecture, as this is what it was designed for, but the heterogeneity of the network, and the fact that multiple neurons are computed on each core means that some networks will be more suited to the platform than others; in particular, neural networks often form 'small world' networking topologies, where most of the connections are relatively local, but there are a few long-distance connections. The computation required to simulate each neuron at each time step in the simulation is generally fixed. The remaining time is then dedicated to processing the spikes received, the number of which depends on the how many neurons are sending spikes to the core and the activity of those connected neurons. This is not known in advance in general, so some flexibility in the system with respect to the amount of computation available at each node is necessary to allow the application to work in different circumstances. Once this is known for a given network, the system could potentially be reconfigured with additional cores, allowing that network to be simulated in less time overall.

4.9.1 PyNN

PyNN is a Python interface to define SNN simulations for a range of simulator back-ends [44]. It allows users to specify an SNN simulation via a Python script once and have it executed on any or all of the supported back-ends including NEST [76], NEURON [33] and Brian [82]. This encourages standardisation of simulators and reproducibility of results, and increases productivity of neural network modellers through code sharing and reuse, by providing a foundation for simulator-agnostic post-processing, visualisation and data-management tools.

PyNN has continued development as part of the European Flagship Human Brain Project (HBP) [4], and has hence been adopted as a modelling language by a number of partners including SpiNNaker. It provides a structured interface for the definition of neurons, synapses and input sources, giving users the flexibility to build a range of network topologies. Models typically consist of single-compartment point neurons, grouped together in *populations*. These populations are then linked with *projections*, representing the synaptic connections between the axons of neurons in a source population, and the dendrites of neurons in a target population. Once defined, a number of simulation controls are used to execute the model for a given time period, with the option to update parameters and initialise state variables between runs. On simulation completion, data can be extracted for post-processing and future reference. Neuron variables such as spike trains, total synaptic conductances and neuron membrane potential are accessible from population objects, while synaptic weights and delays are extracted from projections. These data can be subsequently saved or visualised using the built-in plotting functionality.

Example PyNN commands for the generation of populations and projections are detailed in Listing 4.1. Here the sPyNNaker version of the simulator is imported as *sim* and subsequently used to construct and execute a simulation. A population of 250 Poisson source neurons is created with label ‘poisson_source’ and provides 50 Hz input to the network for 5 s. A second population of 500 integrate and fire neurons is then created and labelled as ‘excitatory_pop’. Excitatory connections are made between ‘poisson_source’ and ‘excitatory_pop’ with a 20% probability of connection, each with a weight of 0.06 nA and delays specified via a probability distribution. Data recording is then enabled for ‘excitatory_pop’, and the simulation is executed for 5 s. Finally, the ‘excitatory_pop’ spike history data are extracted from the simulator.

```
import pyNN.spiNNaker as sim
# Spike input
poisson_spike_source = sim.Population(250, sim.SpikeSourcePoisson(
    rate=50, duration=5000), label='poisson_source')
# Neuronal populations
pop_exc = sim.Population(500, sim.IF_curr_exp(**cell_params_exc),
                         label='excitatory_pop')
# Poisson source projections
poisson_projection_exc = sim.Projection(poisson_spike_source, pop_exc,
    sim.FixedProbabilityConnector(p_connect=0.2),
    synapse_type=sim.StaticSynapse(weight=0.06, delay=delay_distribution),
    receptor_type='excitatory')
# Specify output recording
pop_exc.record('all')
# Run simulation
sim.run(simtime=5000)
# Extract results data
exc_data = pop_exc.get_data('spikes')
```

Listing 4.1. Example PyNN commands (a complete script is detailed in Listing 4.2).

The job of a PyNN simulator is therefore to provide a back-end-specific implementation of the PyNN language, enabling execution of simulations defined in model scripts such as Listing 4.2.

4.9.2 sPyNNaker Implementation

The sPyNNaker Application Programming Interface (API) is comprised of two software stacks as shown in Figure 4.14: one running on host predominantly written in Python, the other running on the SpiNNaker machine written in C.

4.9.3 Preprocessing

At the top of the left-hand side stack in Figure 4.14, users create a PyNN script defining an SNN. The SpiNNaker back-end is specified, which translates the SNN into a form suitable for execution on a SpiNNaker machine. This process includes mapping of the SNN into an application graph, partitioning into a machine graph, generation of the required routeing information and loading of data and applications to a SpiNNaker machine. Once loading is complete, all core applications are instructed to begin execution and run for a predefined period. On simulation completion, requested output data are extracted from the machine and made accessible through the PyNN API.

A sample SNN is developed as a vehicle by which to describe the stages of preprocessing. A random balanced network is defined according to the PyNN script detailed in Listing 4.2, with the resulting network topology shown in Figure 4.15(a). The network consists of 500 excitatory and 125 inhibitory neurons, which make excitatory and inhibitory projections to one another, respectively. Each population additionally makes recurrent connections to itself with the same effect. Excitatory Poisson-distributed input is included to represent background

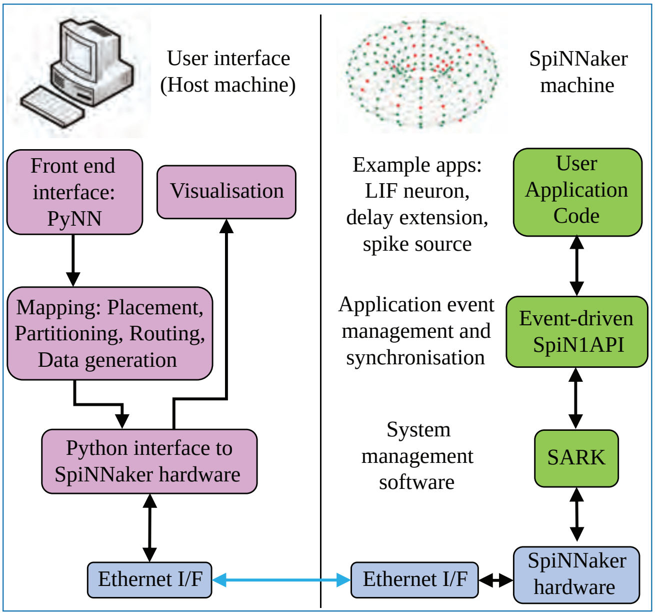

User interface (Host machine)
SpiNNaker machine
Front end interface: PyNN
Visualisation
Mapping: Placement, Partitioning, Routing, Data generation
Python interface to SpiNNaker hardware
Ethernet I/F
Example apps: LIF neuron, delay extension, spike source
User Application Code
Application event management and synchronisation
Event-driven SpiN1API
System management software
SARK
Ethernet I/F
SpiNNaker hardware

Figure 4.14. SpiNNaker software stacks. From top left anti-clockwise to top right: users create SNN models on host via the PyNN interface; the sPyNNaker Python software stack then translates the SNN model into a form suitable for a SpiNNaker machine and loads the appropriate data to SpiNNaker memory via Ethernet; sPyNNaker applications, built on the SARK system management and SpiN1API event-driven processing libraries, use the loaded data to perform real-time simulation of neurons and synapses.

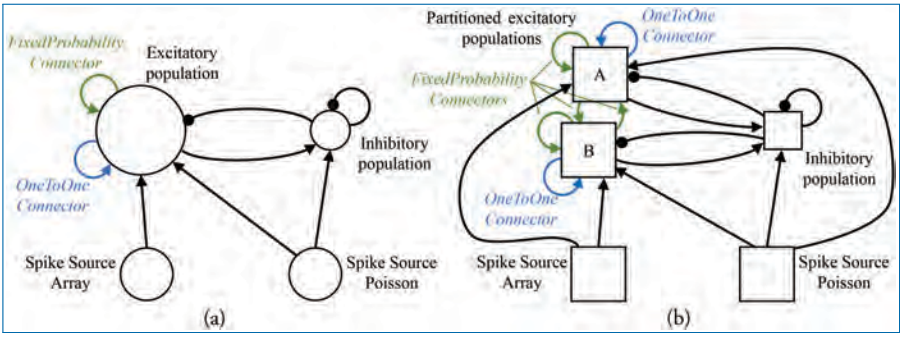


Figure 4.15. Network partitioning to fit machine resources. (a) Application graph generated from interpretation of PyNN script: circles represent PyNN populations, and arrows represent PyNN projections. (b) Machine graph partitioned into vertices and edges to suit machine resources: squares represent populations (or partitioned sub-populations) of neurons which fit on a single SpiNNaker core – hence, the model described by the machine graph in (b) requires 5 SpiNNaker cores for execution.

activity, while predefined spike patterns are injected via a spike source array. The neuronal populations consist of current-based Leaky Integrate and Fire (LIF) neurons, with the membrane potential of each neuron in the excitatory population initialised via a uniform distribution bounded by the threshold and resting potentials. The sPyNNaker API first interprets the PyNN defined network to construct an application graph: a vertices and edges view of the neural network, where each edge corresponds to a projection carrying synapses, and each vertex corresponds to a population of neurons. This application graph is then partitioned into a machine graph, by subdividing application vertices and edges based on available hardware resources and requirement constraints, ultimately ensuring each resulting machine vertex can be executed on a single SpiNNaker core. From hereon, the term *vertex* will refer to a machine vertex and is synonymous with the term sub-population, representing a group of neurons which can be simulated on a single core. An example of this partitioning is shown in Figure 4.15, where due to its size 'excitatory population' is split into two sub-partitions (A and B). Figure 4.15 also shows how additional machine edges are created to preserve network topology between partitions A, B, and the other populations, and how different PyNN connectors are treated differently during this process. For example, a PyNN *OneToOneConnector* connects each neuron in a population to itself. This results in both partitions A and B having a machine edge representing their own connections, but with no edge required to map the connector from one sub-population to the other. Conversely, the PyNN *FixedProbabilityConnector* links neurons in the source and target populations based on connection probability and hence requires machine edges to carry all possible synaptic connections (e.g. both between vertices A and B, and to themselves).

Once partitioned, the machine graph is placed onto a virtual representation of a SpiNNaker machine to facilitate allocation of chip-based resources such as cores and memory. Known failed cores, chips and board links which compromise the performance of a SpiNNaker machine are removed from this virtual representation, and the machine graph is placed accordingly. Chip-specific routeing tables are then generated facilitating transmission of spikes according to the machine edges representing the PyNN-defined projections. These tables are subsequently compressed and loaded into router memory (as described in the previous chapter). The Python software stack from Figure 4.14 then generates the core-specific neuron and synapse data structures and loads them onto the SpiNNaker machine using the SpiNNTools software. Core-specific neuron data are loaded to the appropriate DTCM, while the associated synapse data are loaded into core-specific regions of SDRAM on the same chip, ready for use according to Section 4.9.4. Finally, programs for execution on application cores are loaded to ITCM, with each core executing an initialisation function to load appropriate data structures (from SDRAM) and prepare the core before switching to a *ready* state. Once all simulation cores are *ready*, the signal to begin simulation is given to all cores from host, and the SNN will execute according to the processes defined in Section 4.9.4.

4.9.4 SpiNNaker Runtime Execution

sPyNNaker applications execute SNNs via a hybrid simulation approach, using time-driven neuron updates and event-driven synapse updates, similar to that discussed by Morrison *et al.* [172]. This neuron update scheme provides a flexible framework in which to embed a range of neuron models and is of comparable efficiency to event-based approaches when considering biologically representative spike rates. Synapse events are handled efficiently, with no intermediate information required to update synaptic state between pre-synaptic neuron spikes, which are relatively infrequent on the order of 1 Hz in biological networks. Cores executing sPyNNaker applications hold neuron state variables in local DTCM, allowing efficient access to the required data structures for the periodic time-driven neuron update. Spike transmission between cores is via the AER model [158], with neuronal action potentials communicated as multicast packets, with their key containing only the source neuron ID (in the remainder of this work, the terms: action potential, spike and packet are synonymous). Each packet can be delivered to multiple locations simultaneously via the SpiNNaker routeing fabric, replicating the one-to-many connectivity of an axon. Processing of the packet is performed by the core simulating the post-synaptic neuron, which contains functions to evaluate the spike-based synaptic contribution using only the packet key. Due to the potentially large fan-in to a neuron, memory constraints prevent storage of synaptic data in DTCM. Therefore, the source neuron ID is used to locate the associated synaptic data stored in the relatively large but slower SDRAM memory and copy it locally on spike arrival to facilitate evaluation of the contribution to the synaptic state.

This section focuses on the deployment of this simulation approach within a single core modelling a sub-population of neurons, such as 'Excitatory A' in Figure 4.15(b).

Using the Low-Level Libraries

sPyNNaker applications are compiled against the aforementioned SpiNNaker Application Runtime Kernel (SARK) [251] and the event-driven library SpiN1API [223, 234], as shown in Figure 4.16(a).

In sPyNNaker applications modelling systems of neurons and synapses, callbacks are registered against hardware events: *timer*, *packet received* and *DMA complete*; and a software-triggered *user* event, as shown in Table 4.1. The associated callbacks

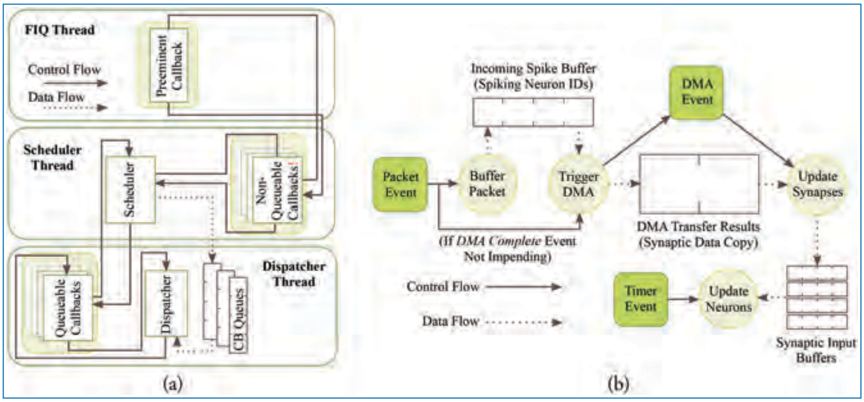


Figure 4.16. SpiNNaker realtime OS: (a) SpiN1API multi-threaded event-based operating system: scheduler thread to queue callbacks; dispatcher thread to execute callbacks; and FIQ thread to service interrupts from high-priority (−1) events. (b) Events and associated callbacks for updating neuron state variables and processing incoming packets representing spikes into synaptic input. Figures reproduced with permission from [222, 223].

Table 4.1. Hardware (and single software) events, along with their registered callback and associated priority level.

Event	Callback	Priority	Pre-empts priority
Packet received	_multicast_packet_received_callback	−1	0, 1, 2
DMA complete	_dma_complete_callback	0	1, 2
Timer	timer_callback	2	–
User (Software)	user_callback	0	1, 2

facilitate the periodic updating of neuron state and the event-based processing of synapses when packets representing spikes arrive at a core. These events (squares) and their callbacks (circles) are shown schematically in Figure 4.16(b). The function timer_callback evolves the state of neurons in time and is called periodically against *timer* events throughout a simulation. A *packet received* event triggers a _multicast_packet_received_callback, which reads the packet to extract and transfer the source neuron ID to a spike queue. If no spike processing is currently being performed, the software-triggered *user* event is issued and, in turn, executes a user_callback that reads the next ID from the spike queue, locates the associated synaptic information stored in SDRAM and initiates a DMA to copy it into DTCM for subsequent processing. Finally, the _dma_complete_callback is executed on a *DMA complete* event and initiates processing of the synaptic contribution(s) to the post-synaptic neuron(s). If on completion of this processing there are items remaining in the input spike queue, this callback initiates processing of the next spike: meaning this collection of callbacks can be thought of as a spike processing pipeline.

Time-Driven Neuron Update

A sPyNNaker simulation typically contains multiple cores, each simulating a different population of neurons (see Figure 4.15(b)). Each core updates the states of its neurons in time via an explicit update scheme with fixed simulation timestep (Δt). When a neuron is deemed to have fired, packets are delivered to all cores that neuron projects to and processed in real time by the post-synaptic core to evaluate the resulting synaptic contribution. Therefore, while all cores operate asynchronously, it is desirable to advance neurons on all cores approximately in parallel to march forward a simulation coherently. All cores in a simulation therefore start synchronised and register *timer* events with common frequency, with the period between events defined by a fixed number of clock cycles, as shown in Figure 4.17. All cores will therefore initiate a *timer* event and execute a timer_callback to advance the state of their neurons approximately in parallel, although the system is asynchronous as there is no hardware or software mechanism to synchronise cores. Individual update times may vary due to any additional spike processing (see Section 4.9.4); however, cores that have additional spikes to process between one pair of timer events can catch up during subsequent periods of lower activity. Relative drift between boards

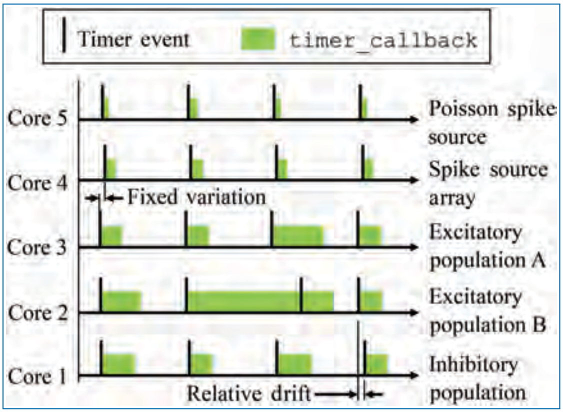


Figure 4.17. Time-driven updates by neuron cores simulating the network in Figure 4.15(b): periodic *timer* events trigger callbacks advancing neuron states by Δt. Cores can be out of phase due to communication of the start signal, and relative drift can occur due to manufacturing variability between boards. Note that state update times vary with the level of additional spike processing within a simulation timestep, however cores which experience high levels of spike activity delaying the subsequent time_callback can catch up during subsequent periods of lower spike activity (as shown by Core 2).

is possible due to slight variations in clock speed (from clock crystal manufacturing variability); however, this effect is small relative to simulation times [235]. Small variations placing core updates slightly out of phase can also occur due to the way the 'start' signal is communicated, particularly on larger machines; however, again this effect is negligible. A consequence of this update scheme is that generated spikes are constrained to the time grid (multiples of the simulation timestep Δt). It also enforces a finite minimum simulation spike transit time between neurons of Δt, as input cannot be guaranteed to arrive in the current timestep before a neuron has been updated. From the hardware perspective, the maximum packet transit time for the million core machine is $\leq 25\,\mu$s (assuming 200 ns per router [235], and a maximum path length of 128).

A design goal of the SpiNNaker platform is to achieve real-time simulation of SNNs, where 'real time' is defined as when the time taken to simulate a network matches the amount of time the network has modelled. Therefore, an SNN with a simulation timestep of $\Delta t = 1$ ms requires the period of *timer* events to be set at 200,000 clock cycles (where at 200 MHz each clock cycle has a period of 5 ns – see Section 2.2). This causes 1 ms of simulation to be executed in 1 ms, meaning the solution will keep up with wall-clock time, enabling advantageous performance, and interaction with systems operating on the same clock (such as robots, humans and animals). In practice, real-time execution is not always possible, and therefore, users are free to reduce the value of Δt in special cases and also adjust the number of clock cycles between *timer* events. For example, if a neuron model requires $\Delta t = 0.1$ ms for accuracy, it is a common practice to let the period between *timer* events remain at 200,000 clock cycles, to ensure there is sufficient processing time to update the neurons and process incoming spikes [217]. This enforces a slowdown factor of 10 relative to real time.

From the perspective of an individual core, each neuron is initialised with user-defined parameters at time t_0 (supplied via a PyNN script). All state variables are then updated one timestep at a time up to the simulation end time t_{end}. The number of required updates and hence *timer* events is calculated based on t_{end} and the user-defined simulation timestep Δt (which is fixed for the duration of simulation). Each call to timer_callback advances all the neurons on a core by Δt according to Algorithm S1 in [208], which is shown schematically on the left-hand side of Figure 4.18. First the synapse state for all neurons on the core is updated according to the model shaping rule, and any new input this timestep is added from the synaptic input buffers (discussed below). Interrupts are disabled during this update to prevent concurrent access to the buffers from spike processing operations. The states of all neurons on the core are then updated sequentially. An individual neuron state at the current time $N_{i,t}$ is accessed in memory, and if the neuron is not refractory, its state is updated according to the model characterising its sub-threshold

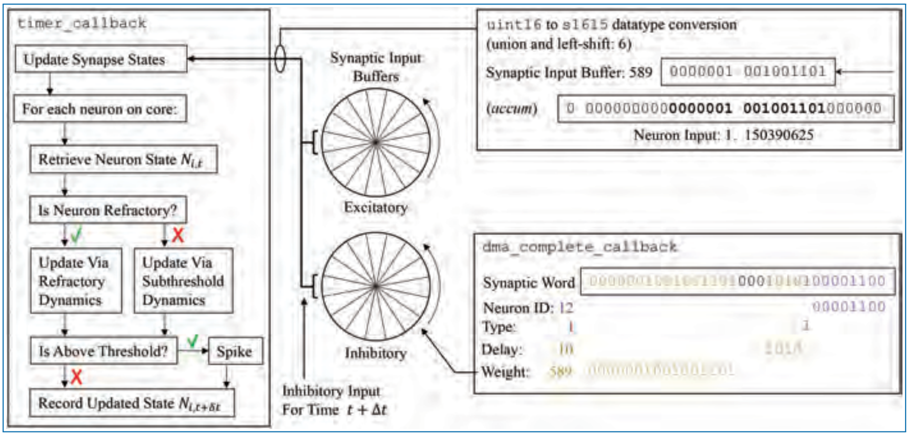


Figure 4.18. Left: update flow advancing state of neuron N_i by Δt. Centre: circular synaptic input buffers accumulate scaled input at different locations based on synaptic delay (buffers are rotated one slot at the end of every timestep). Right top: synaptic input buffer values are converted to fixed-point format and scaled before adding to N_i. Right bottom: decoding of synaptic word into circular synaptic buffer input.

dynamics (see examples in Section 4.9.5). If it is judged to have emitted a spike, the refractory dynamics are initiated and the router is instructed to send a multicast packet to the network. Finally, all requested neuron variables are recorded as belonging to this new timestep ($t + \Delta t$) and stored in core memory for subsequent extraction by the SpiNNTools software – interrupts are disabled during this process to prevent concurrent access to recording datastructures.

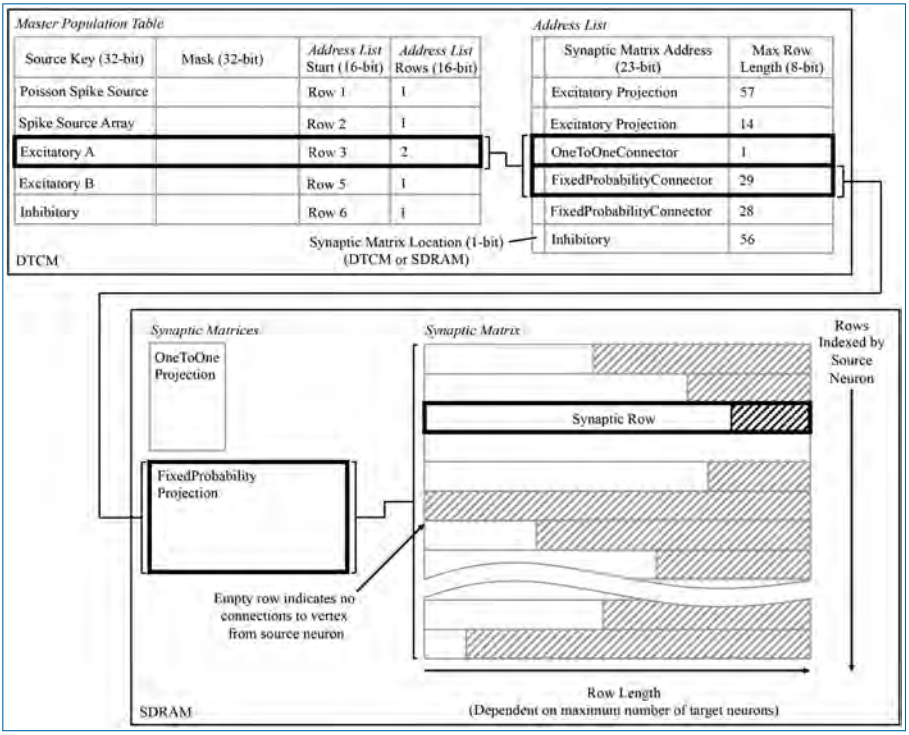
Synaptic input buffers (Figure 4.18 centre) are used to accumulate all synaptic input on a given receptor type, removing the computational cost of managing state variables for individual synapses (as developed by Morrison *et al.* [172]). Each buffer is constructed from a number of 'slots', where each slot represents input at a future simulation timestep. All input designated to arrive at a particular time is accumulated in the appropriate slot, constraining synapse models to those whose contributions can be summed linearly. A pointer is maintained to the input associated with the proceeding timestep ($t + \Delta t$). Each neuron update consumes the input addressed by this pointer and then advances it forward one slot (effectively rotating the buffer). When the pointer reaches the last slot, it cycles back to the first, meaning these slots continuously represent input over the next d timesteps, where d is the number of slots. By default the value of d is set via a 4-bit unsigned integer, enabling representation of delays up to 16 timesteps (however, Section 4.9.6 contains information on extending this delay). In the default sPyNNaker implementation, a synaptic input buffer is created per neuron, per receptor type, and is a collection of 16 slots each constructed from unsigned 16-bit integers. The use of an integer representation reduces buffer size in DTCM and also the size of synaptic weights in SDRAM, relative to using standard 32-bit fixed-point *accum* type. However, it requires conversion to *accum* type for use in the neuron model calculations – as shown in Figure 4.18. This conversion is performed via a union and left-shift, the size of which represents a trade-off between headroom and precision. An example shift of 6 is shown, causing the smallest bit of the synaptic input buffer to represent $2^{-9} = 1.953125 \times 10^{-3}$, and the largest $2^7 = 128$, in the *accum* type of the synapse state. Under extreme conditions, a buffer slot will saturate from concurrent spike activity, meaning the shift size should be increased. However, the shift is also intrinsic to the weight representation and affects precision, as all weights must be scaled by $2^{(15-shift)}$ before being written as integers to the synaptic matrices discussed in Section 4.9.4. For example, in Figure 4.18, a weight of 1.15 nA was converted to 589 on host during generation of synaptic data, but is returned as 1.150390625 nA when used during simulation (with a shift of 6). The shift value is currently calculated by the sPyNNaker toolchain to provide a balance between handling large weights, high fan-in and/or pre-synaptic firing rates, and maintaining precision – see the work by Albada *et al.* [3] where the theory leading to a usable closed-form probabilistic headroom mechanism is described in Equation 1.

Receiving a Spike

A _multicast_packet_received_callback is triggered by a *packet received* event, raised when a multicast packet arrives at the core. This callback is assigned highest priority (−1) and hence makes use of the FIQ thread and pre-empts all other core processing (see Figure 4.16(a)). This callback cannot be queued, and therefore, to prevent traffic backing up on the network, this callback is designed to execute quickly, and it simply extracts the source neuron ID (from the 32-bit key) and stores it in an input spike buffer for subsequent processing. Note that by default this buffer is 256 entries long, enabling queuing of 256 spikes simultaneously. The callback then checks for activity in the spike processing pipeline and registers a *user* event if inactive. Pseudo code for this callback is made available by Rhodes *et al.* [208].

Activation of the Spike Processing Pipeline

A user_callback callback is triggered by the *user* event registered in a Section 4.9.4 and kick-starts the spike processing pipeline. The callback locates in SDRAM the synaptic data associated with the spike ID and initiates its DMA transfer to DTCM for subsequent processing. Three core-specific data structures are used in this process: the *master population table*, *address list* and *synaptic matrix*. Use of these data structures is shown schematically in Figure 4.19, from the perspective of the core simulating the Excitatory A population in Figure 4.15(b), when receiving a spike from the Excitatory A population. The *master population table* is a lightweight list



Master Population Table

Source Key (32-bit)	Mask (32-bit)	Address List Start (16-bit)	Address List Rows (16-bit)
Poisson Spike Source		Row 1	1
Spike Source Array		Row 2	1
Excitatory A		Row 3	2
Excitatory B		Row 5	1
Inhibitory		Row 6	1

Address List

	Synaptic Matrix Address (23-bit)	Max Row Length (8-bit)
	Excitatory Projection	57
	Excitatory Projection	14
	OneToOneConnector	1
	FixedProbabilityConnector	29
	FixedProbabilityConnector	28
	Inhibitory	56

Synaptic Matrix Location (1-bit)
(DTCM or SDRAM)
DTCM
Synaptic Matrices
OneToOne Projection
FixedProbability Projection
Empty row indicates no connections to vertex from source neuron
Synaptic Matrix
Synaptic Row
Rows Indexed by Source Neuron
Row Length
(Dependent on maximum number of target neurons)
SDRAM

Figure 4.19. Data structures for processing incoming spikes: *Master population table*, *address list*, and *synaptic matrix*, are shown from the perspective of the core simulating the Excitatory A population in Figure 4.15(b). The path in bold represents that taken when a packet is received by Excitatory A, originating from itself, and hence two projections must be processed.

taking a masked source neuron ID as the key by which a source vertex can be identified. Each row pertains to a single source vertex and consists of: 32-bit key; 32-bit mask; 16-bit start location of the first row in the *address list* pertaining to this source vertex; and a 16-bit value defining the number of rows, where each row in the *address list* represents a PyNN projection. When searching this table, the key from the incoming packet is masked using each entry-specific mask before comparing to the entry key. This masks off the individual neuron ID bits and enables source vertices to simulate different numbers of neurons. The entry keys are masked on host before loading for efficiency and are structured to prevent overlap after masking and facilitate binary searching. The structure of an *address list* row consists of: a single header bit detailing whether the synaptic matrix associated with this projection is located in DTCM or SDRAM; 32-bit memory address indicating the first row of the synaptic matrix; and an 8-bit value detailing the synaptic matrix row length (i.e. the maximum number of post-synaptic neurons connected to by a pre-synaptic neuron in a particular projection). Note that synaptic matrix rows are indexed by source neuron ID and that all rows are padded to the maximum row length to facilitate retrieval, including empty rows for pre-synaptic neurons not connected to neurons on this core. The row data structure is covered in detail in Section 4.9.4.

This callback therefore takes from the input spike buffer the next spike ID to process and uses it in a binary search of the *master population table* to locate the *address list* regions capturing the projections carrying the spike to this vertex. The SDRAM location and size specified by each row are then used in sequential processing of the projections. For the case shown in Figure 4.19, searching the *master population table* yields two rows in the *address list*, which in turn define the location of the corresponding synaptic matrices in SDRAM. Each synaptic matrix is indexed according to pre-synaptic neuron ID, enabling location of the appropriate row to copy to core DTCM for processing of each spike. Details of this row are then passed to the DMA controller to begin the data transfer, marking the end of the callback. This allows the core to return to processing other callbacks, hiding the DMA transfer as shown for 'Spike 1' in Figure 4.21.

Synapse processing

On completion of the DMA in Section 4.9.4, a *DMA complete* event triggers a _dma_complete_callback, initiating processing of the synaptic row. As described previously, each row pertains to synapses made, within a single PyNN projection, between a single pre-synaptic neuron and multiple post-synaptic neurons. At the highest level, a synaptic row is an array of synaptic words, where each word is defined as a 32-bit unsigned integer. The row is split into three designated regions to enable identification of static and plastic synapses (connections capable of changing their weight at runtime). The row regions contain dynamic plastic data, constant fixed plastic data and static data. Three header fields are also included, detailing the size of each region and enabling easy navigation of the row. A schematic breakdown of the synaptic row structure is detailed in Figure 4.20. Note that because a PyNN projection cannot be both static and plastic simultaneously, a single row contains only either static or plastic data. Plastic data are intentionally segregated into dynamic and fixed regions to facilitate processing. While all plastic data must be copied locally to evaluate synaptic contributions to a neuron, only the dynamic region – that is, that changing at runtime – requires updating for use when processing subsequent spikes. Keeping this dynamic data in a separate block facilitates writing back to the synaptic matrix with a single DMA, and writing back less data helps compensate for reduced DMA write bandwidth (relative to read – see Section 2.2).

The static region occupies the lower portion of the synaptic row and is itself an array of synaptic words, where each word corresponds to a synaptic connection

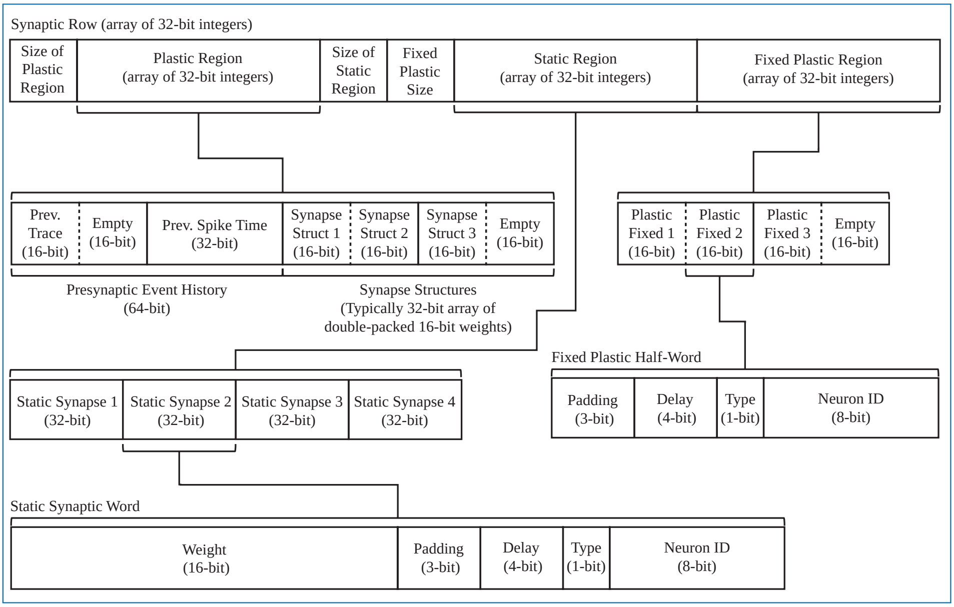

Synaptic Row (array of 32-bit integers)
Size of Plastic Region
Plastic Region (array of 32-bit integers)
Size of Static Region
Fixed Plastic Size
Static Region (array of 32-bit integers)
Fixed Plastic Region (array of 32-bit integers)
Prev. Trace (16-bit)
Empty (16-bit)
Prev. Spike Time (32-bit)
Synapse Struct 1 (16-bit)
Synapse Struct 2 (16-bit)
Synapse Struct 3 (16-bit)
Empty (16-bit)
Presynaptic Event History (64-bit)
Synapse Structures (Typically 32-bit array of double-packed 16-bit weights)
Plastic Fixed 1 (16-bit)
Plastic Fixed 2 (16-bit)
Plastic Fixed 3 (16-bit)
Empty (16-bit)
Fixed Plastic Half-Word
Padding (3-bit)
Delay (4-bit)
Type (1-bit)
Neuron ID (8-bit)
Static Synapse 1 (32-bit)
Static Synapse 2 (32-bit)
Static Synapse 3 (32-bit)
Static Synapse 4 (32-bit)
Static Synaptic Word
Weight (16-bit)
Padding (3-bit)
Delay (4-bit)
Type (1-bit)
Neuron ID (8-bit)

Figure 4.20. Synaptic row structure with breakdown of substructures for both static and plastic synapses.

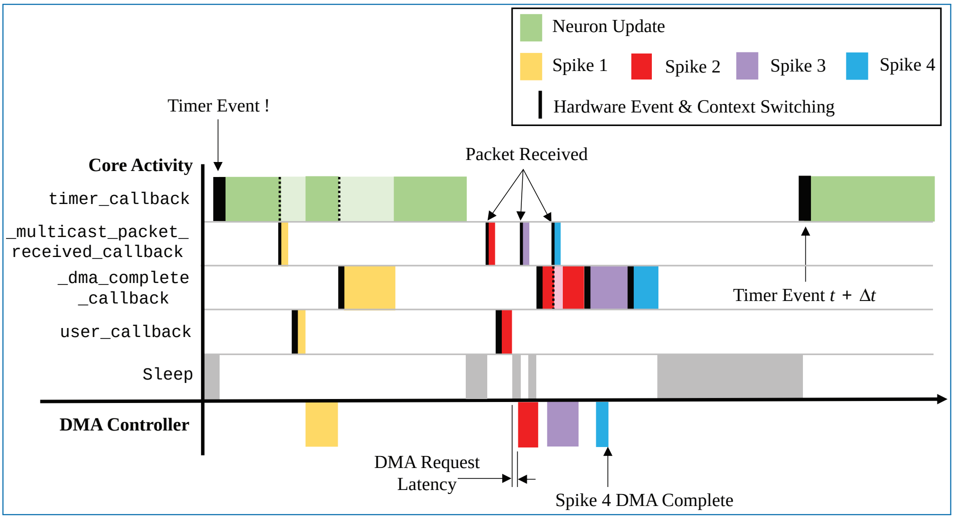


Figure 4.21. Interaction of callbacks shown over the time period between two *timer* events. Four spike events are processed representing the scenarios: receiving a packet while processing a timer event; receiving a packet while the core is idling; and receiving a packet while the spike processing pipeline is active. Note that a lighter colour shade indicates a suspension of a callback, which is resumed on completion of higher priority tasks.

between the row's pre-synaptic neuron and a single post-synaptic neuron. As shown in Figure 4.20, each 32-bit data structure is split such that the top 16 bits represent the weight, while the lower 16 bits typically split: bottom 8 bits to specify the post-synaptic neuron ID; 1 bit to specify the synapse type (excitatory 0, or inhibitory 1); 4 bits to specify synaptic delay; leaving 3 bits for padding (useful for model customisation, e.g., adding additional receptors types). Data defining plastic synapses are divided across the dynamic and fixed regions. Fixed plastic data are defined by a 16-bit unsigned integer and match the structure of the lower half of a static synapse (see lower half of Figure 4.20). These 16-bit synaptic half-words enable double-packing inside the 32-bit array of the synaptic row, meaning an empty half-slot will be apparent if the row targets an odd number of synapses. The dynamic plastic region contains a header defining the *pre-synaptic event history*, followed by a series of *synapse structures* capturing the weight of each synapse. Note that for typical plasticity models, this defaults to the same 16-bit weight describing static synapses; however, *synapse structure* can be extended to include additional parameters (in multiples of 16 bits) if required by a given plasticity rule.

A task of the _dma_complete_callback is therefore to convert the synaptic row into individual post-synaptic neuron input. The callback processes the row headers to ascertain whether it contains static or plastic data, adjusts synapses according to a given plasticity rule, and then loops over each synaptic word and extracts its neuronal contribution – pseudo code for this callback is detailed in Algorithm S4 of [208]. An example of this process for a single static synaptic word is shown in the lower right of Figure 4.18, where a synaptic word of [0000001001001101 0001010100001100] leads to a contribution of 589 to slot 10 of the inhibitory synaptic input buffer for neuron N_{12}.

Callback Interaction

The callbacks described above define how a sPyNNaker application responds to hardware events and updates an SNN simulation. The interaction of these events is a complex process, with the potential to impact the ability of a SpiNNaker machine to perform real-time execution. Figure 4.21 covers the time between two *timer* events and shows interaction of spike processing and neuron update callbacks for four scenarios detailed by the arrival of spikes 1–4. The first *timer* event initiates processing of the neuron update; however, after completion of approximately one-third of the update, the core receives Spike 1, interrupting the timer_callback and triggering execution of a _multicast_packet_received_callback, which in turn raises a *user* event, initiating DMA transfer of the appropriate synaptic information. On completion of the callback, the core returns to the timer_callback, with the DMA transfer occurring in parallel. On completion of the DMA, a _dma_complete_callback is initiated, which processes the transferred synaptic information into neuronal input. The core then returns to the timer_callback, which continues to completion. The core is idle when it receives Spike 2; therefore, processing of the spike begins immediately, and the subsequent *user* event and hence DMA request is initiated. While waiting for the data to transfer, Spike 3 is received, and the associated _multicast_packet_received_callback is processed. This time, due to the active spike processing pipeline, no *user* event is raised, and instead, the DMA for Spike 3 is initiated at the beginning of the _dma_complete_callback triggered by Spike 2. Whilst processing this callback, Spike 4 is received, and the associated _multicast_packet_received_callback interrupts the core to place the packet key in the input spike queue. This queue entry is eventually processed at the beginning of the _dma_complete_callback for Spike 3, demonstrating the spike processing pipeline in action. This also shows the benefit of having two hardware 'threads' working in parallel, as the core is utilised completely, and the DMA transfer is hidden behind the _dma_complete_callback, when the pipeline is active. Finally, after an idle period (where the processor is put to sleep in a low energy state), the next *timer* event is issued at time $t + \Delta t$.

From Figure 4.21, it is seen that core processing is dependent on SNN activity. When targeting real-time execution (Section 4.9.4), it is important to consider extreme circumstances and how they will affect both the core and global simulation. For example, it is clear from Figure 4.21 that when a core receives spikes, it can delay completion of the timer_callback due to the assigned callback priorities (as shown in Figure 4.17). This is a design choice, as it helps maximise core utilisation by hiding DMA transfers behind the timer_callback when the spike processing pipeline is inactive. However, in the extreme case, spike processing will delay the completion of the callback beyond the issuing of the next *timer* event. While the core can potentially catch up this lost time, this scenario has the potential to delay the neuron update beyond a single *timer* event and ultimately cause any spike packets emitted from this core to be received and processed at the wrong time by the rest of the network. To guard against this, sPyNNaker applications report any occurrences of an overrun, where a timer_callback is not complete before the next *timer* event is raised and also the maximum number of *timer* events that a single timer_callback overruns. Similar metrics are also reported when the input spike queue overflows (exceeds 256 entries) and when the synaptic input buffers saturate. Together these metrics provide a window into the ability of a core to handle the required processing within a simulation.

Another important performance consideration when responding to spike packets using prioritised events is the time taken to switch between the associated callbacks. Events are displayed in Figure 4.21 by solid black lines, the width of which represents the time taken to switch context and begin execution of the callback. The timer_callback takes longest to respond due to queuing of events with priority > 0, while the _multicast_packet_received_callback is quickest due to its priority of -1 and use of the FIQ thread. Other chip-level factors can also influence execution, such as SDRAM contention with applications running on adjacent cores. As DMAs are processed in serial bursts, if multiple simultaneous requests are received by the SDRAM controller, there may be latency in beginning the DMA for some cores and a reduced rate of transfer (see Section S1.2 of [208] for further information).

4.9.5 Neural Modelling

At the heart of a sPyNNaker application is the solution of a series of mathematical models governing neural dynamics. It is these models which determine how incoming spikes affect a neuron and when a neuron itself reaches threshold. While the preceding section described the underlying event-based operating system facilitating simulation and interaction of neurons, this section focuses on the solution of equations governing neural state and how they are structured in software.

Software Structure

PyNN defines a number of standard cell models, such as the LIF neuron and the Izhikevich neuron. Implementations of these standard models are included in sPyNNaker; however, the API is also designed to support users wishing to extend this core functionality and implement neuron models of their own. To facilitate this extension, the model framework is defined in an object-oriented fashion, through the use of C code on the SpiNNaker machine. This modular approach provides structure and aids code reuse between different models (e.g. sharing of a synaptic plasticity rule between different neuron models). A neuron model is built from the following components:

- *synapse_type*, defining how synapse state evolves between pre-synaptic spikes and how contributions from new spikes are added to the model. A fundamental requirement is that multiple synaptic inputs can be summed and shaped linearly, such as the α-kernel [49].
- *neuron_model*, implementing the sub-threshold update scheme and refractory dynamics.
- *input_type*, governing the process of converting synaptic input into neuron input current. Examples include current-based and conductance-based formulations [45].

- *threshold_type*, defining a system against which a neuron membrane potential is compared to adjudge whether a neuron has emitted a spike.
- *additional_input_type*, offering a flexible framework to model intrinsic currents dependent on the instantaneous membrane potential and potentially responding discontinuously on neuron firing (such as the Ca^{2+}-activated K^{+} current described by Liu and Wang [149]).

The individual model components each produces a subset of the neuron and synapse dynamics and is therefore the entry point for a user looking to deploy a custom neuron model.[4] In keeping with the aforementioned software stacks in Figure 4.14, interfaces to each component are written in both Python and C. A single instance of each component is collected via a C header file and compiled against the underlying operating system described in Section 4.9.4 to generate a runtime application. Python classes for each component facilitate user interaction with each part of the model, enabling setting of parameter values and initial conditions from a PyNN SNN script.

The runtime execution framework calls each component as part of the timer_callback, as detailed in Algorithm S1 in [208] and shown schematically in Figure 4.18. First the synaptic state is advanced forward in time by a single simulation timestep, using the functions defined by the *synapse_type* component. Core interrupts are disabled during this process to prevent concurrent access of the synaptic input buffers from a _dma_complete_callback. Interrupts are re-enabled when all the state related to the synapses for all receptor types for all neurons on a core have been updated. Each neuron then has its state advanced by Δt. The *input_type* component is called first, converting the updated synaptic state into neuron input current. This includes separate excitatory and inhibitory components, with core implementations capable of handling both current- and conductance-based formulations. The *additional_input* component is then evaluated to calculate the level of any intrinsic currents. The synaptic and intrinsic currents, together with any background current, are then supplied to the *neuron_model* component which subsequently marches forward the neuron state by Δt. The neuron membrane potential is now passed to the *threshold_type* component which tests whether the neuron has fired. If the neuron is above threshold, a number of actions are performed: a refractory counter begins to instigate any refractory period; the *additional_input* is notified of the spike to allow updating of appropriate state variables; and finally, the core is instructed to send a multicast packet to the router with the neuron ID as key.

[4] A detailed guide to this process can be found at: http://spinnakermanchester.github.io/workshops/seventh.html

Leaky Integrate and Fire Neuron

The sPyNNaker implementation of a current-based LIF neuron is described by the hybrid system in Equations 4.1 and 4.2. The sub-threshold dynamics are governed according to Equation 4.1, where V is the membrane potential, I is the input current (combining synaptic, intrinsic and background input), R_m is the membrane resistance, τ_m is the membrane leak time constant and E_l is the membrane leak (resting) potential.

$$\frac{dV}{dt} = -\frac{V - (E_l + R_m I(t))}{\tau_m} \qquad \text{if } V > V_\theta, \; V = V_{reset} \tag{4.1}$$

$$\frac{dI_{syn}}{dt} = -\frac{I_{syn}}{\tau_{syn}} + \delta(t - t^j) \tag{4.2}$$

If V exceeds the threshold level V_θ, the neuron is reported to have spiked and V is set to the reset potential V_{reset} for the refractory period duration t_r. Synaptic currents I_{syn} are modelled according to Equation 4.2, where τ_{syn} is the synaptic time constant (independent value for each receptor type), and the delta function represents addition of a step change in input from the weight of an incoming spike.

The sPyNNaker implementation embeds Equation 4.2 in a *synapse type* component, providing mechanisms to update the input current both between spikes (i.e. when the synaptic input buffer contribution is zero) and on spike arrival. Exact integration is used to update the synapse state during the periodic neuron update, with step changes made from synaptic input buffer contributions according to Equation 4.3.

$$I_{t+1} = I_t e^{-\Delta t/\tau_{syn}} + \Sigma_j w_{ij} \delta(t - t_j) \tag{4.3}$$

The constant factor $e^{-\Delta t/\tau_{syn}}$ is pre-calculated before loading to the SpiNNaker machine to avoid evaluation at runtime, as both the divide and exponential operations are relatively expensive on the ARM968 (≈100 clock cycles each). A *neuron model* component captures the neuron state update mechanism, which solves Equation 4.1 via exponential integration [212] and assuming the change in current over the timestep is small [45], yielding the update function in Equation 4.4.

$$V_{t+1} = E_l + R_m I_{t+\Delta t} - e^{-\frac{\Delta t}{\tau_m}} (E_l + R_m I_{t+\Delta t} - V_t) \tag{4.4}$$

To compensate for this assumption, w_{ij} is decayed before adding to the synapse to ensure the total charge input to a neuron matches the exact solution [3]. Static thresholding defined via the *threshold type* compares the instantaneous membrane potential to the threshold level V_θ.

Izhikevich Neuron

The Izhikevich neuron model [116] allows reproduction of biologically observed neuronal characteristics such as spiking and bursting. Its dynamics follow a type of 'quadratic integrate and fire' model, as detailed in Equation 4.5

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I(t) \tag{4.5}$$
$$\frac{du}{dt} = a(bv - u)$$

$$\text{if } v \geq V_\theta, \text{ then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{4.6}$$

where v and u are dimensionless variables representing the membrane potential and a recovery variable, respectively. Dimensionless parameters a, b, c and d are used to tune the model dynamics, and I represents combined background, intrinsic and synaptic currents. If v exceeds a threshold V_θ, v and u are reset according to Equation 4.6.

The sPyNNaker implementation of this model uses the same *synapse type*, current-based *input type* and static *threshold type* components as the aforementioned LIF implementation. However, updating the neuron state and hence solving the system defined by Equation 4.5 requires numerical integration. A range of solvers were explored with fixed-point data type by Hopkins and Furber [101], with the RK-2 midpoint preferred as a trade-off between speed and accuracy. The resulting explicit update scheme is detailed in Equation 4.7.

$$\theta = 140 + I_{t+\Delta t} - u_t \qquad \alpha = \theta + (5 + 0.04v_t)v_t$$
$$\eta = \frac{\alpha h}{2} + v_t \qquad \beta = \frac{ah}{2}(bv_t - u_t)$$
$$v_{t+\Delta t} = v_t + h(\theta - \beta + (0.04\eta + 5)\eta)$$
$$u_{t+\Delta t} = u_t + ah(b\eta - \beta - u_t) \tag{4.7}$$

While it is hard to recognise the original equations in this form, refactoring of the update scheme and algebraic manipulation leads to several improvements in the implementation. The use of intermediate variables not only enables compiler optimisations improving speed and code size but also helps prevent over/underflow of the *accum* data type during intermediate calculations [101].

4.9.6 Auxiliary Application Code

While neuron-simulating applications capture the core operations of an SNN, several additional sPyNNaker applications are required to generate network input and facilitate network operation. These single-core applications are built following similar principles to those defined in Section 4.9.4, responding via the same event-based operating system to send and receive packets and interact with neuron cores. They are embedded in the machine graph during network preprocessing and loaded onto a SpiNNaker machine together with configuration data.

Spike Input Generation

Generating spikes is an integral part of SNN simulations. It enables modelling of network response to specific patterns of spikes and input representing adjacent brain regions or background noise. The sPyNNaker API includes two applications for spike generation: *spike source array* and *Poisson spike source*. These applications are built from compiled C and require a single SpiNNaker core per instance. They follow *timer* events in parallel (but asynchronously) with neuron-simulating cores and send multicast packets representing spikes as discussed previously. These applications do not receive spikes and hence have their functionality encoded entirely in callbacks registered against *timer* events. As with all sPyNNaker applications, a corresponding Python class enables construction of a spike generator in a PyNN script and allows configuration data to be specified and subsequently loaded to a SpiNNaker machine.

The *spike source array* application contains a population of neuron-like units which emit spikes at specific times (see Listing 4.2). The times and keys to emit are stored in SDRAM and only copied into local DTCM when required during execution. The buffer of times/keys is pre-loaded up to memory limits and can be replenished during execution by sending requests to the host, although this is limited by the bandwidth of the on-board Ethernet. Callbacks issued on *timer* events (corresponding to timestep updates on neuron cores) then send packets to the router at the prescribed times. If multiple 'neurons' are registered to emit spike packets over the same timestep, a small random delay is added between sending of the packets to reduce pressure on the router.

The *Poisson spike source* application emits packets according to a Poisson distribution about a given frequency. A population of neuron-like units is specified, each of which can be assigned an individual mean firing rate (see Listing 4.2). At runtime, periodic *timer* events trigger a callback at every simulation timestep Δt, which assesses whether the core should send a packet to the router representing a spike. A distinction is made between slow and fast Poisson spike sources based on whether they emit fewer >1 spike for any Δt. For fast spike sources, the number of spikes to send between *timer* events is calculated [130], and the corresponding packets sent are interspersed with random delays. This random spacing reduces the chance of synchronised spike arrival at post-synaptic cores, easing pressure on both the source and target routers. For slow sources, after each spike, an inter-spike interval is evaluated in multiples of Δt, which is then counted down between sending packets. For fast spike sources, the post-synaptic core is likely to retrieve from SDRAM the same pieces of synaptic matrix many times during a simulation. Therefore, to remove the overhead of the DMA, a mechanism is included to store the synaptic matrices from fast spike sources in DTCM.

Simulating Extended Synaptic Delays

While there is a mechanism in the synaptic row to account for delays of up to $16\Delta t$, it can be necessary to prescribe longer delays (particularly when Δ is small). To account for this case, an application called a *delay extension* is created [3], running on an adjacent core. Packets representing spikes exhibiting a delay $\geq 16\Delta t$ are routed to the core running this application, which subsequently sends new spikes targeting the post-synaptic core after a sufficient portion of the delay has elapsed such that any remaining delay can be handled within the synaptic row.

Two data structures are used to manage delay handling: a ‘delay stage configuration’ is generated during preprocessing and captures the size of delay associated with each pre-synaptic neuron; and a ‘spike counter’ registers the time and pre-synaptic neuron of incoming spikes. Two callbacks are used in the *delay extension*, registered against *packet received* and *timer* hardware events. On packet arrival, the first callback extracts the pre-synaptic neuron ID to an input spike buffer, similar to the process described in Section 4.9.4. The second callback is executed on *timer* events occurring in parallel (but asynchronously) with those on neuron processing cores. The callback processes any spikes received since the previous *timer* event, taking entries from the input spike buffer and using them to update the spike counting data structure to register the incoming spikes against multiples of the number of synaptic input buffer slots on the corresponding post-synaptic core. There are typically 16 such slots, where in this context a collection of 16 slots is referred to as a ‘delay stage’. A second data structure captures how many delay stages each spike should be held for before being released to the post-synaptic core. Therefore, using these two data structures, it is possible to assess the incoming spikes to calculate the corresponding outgoing spike times and hence schedule the necessary spikes for distribution to the network.

While this application solves the problem of simulating extended delays, it cannot do so indefinitely and an effective new upper limit of $144\Delta t$ is enforced due to DTCM constraints. It should also be noted that this mechanism introduces additional overhead to the system: an extra core is required to run the application, and two packets are now required to transmit a spike. The post-synaptic core also performs additional processing during look-up of the source vertex in the *master population table*. An additional row must be included to identify spikes travelling direct from the pre-synaptic core and also those sent from each individual delay stage of the *delay extension*. This increased *master population table* size can be costly to search and detrimental for real-time performance [207].

4.10 Software Engineering for Future Systems

As this text is being written, the SpiNNaker2 system, described in Chapter 8, is being developed. This architecture has clear implications on the software. To this end, it makes sense to develop the software to require as few changes as possible to make it compatible with this new system. The hierarchical modular structure of the software supports this well; for example, the mapping algorithms operate on a machine object, which can be simply updated when the structure of the new system is known (or a second version can be created and algorithms can operate on whichever system is in use). Similarly, the communications layer will require updating to match the communications used by the new system. However, the concepts will be similar to the higher levels, and so they will be able to stay the same, for example, the communications layer will have to support 'executing of a binary' and 'loading of data' but the signature of these functions can be made the same for both the old and the new system, avoiding the need to change the high-level libraries.

The other part of the system that would require changes is within the C code, where the features of the new system will need to be made accessible through the low-level libraries. Again, concepts that exist in both systems, such as the ability to run in an event-based manner, will map directly to the hardware, and so high-level code will not have to change (other than requiring recompilation of course).

All this means that minimal code changes will be required to make the code compatible with both old and new (and possibly even newer) systems. This makes the code somewhat future proof in so much as any software can be and require minimal maintenance as the hardware systems are developed.

4.11 Full Example Code Listing

```
import pyNN.spiNNaker as sim

# Initialise simulator
sim.setup(timestep=1)

# Spike input
poisson_spike_source = sim.Population(250, sim.SpikeSourcePoisson(
    rate=50, duration=5000), label='poisson_source')

spike_source_array = sim.Population(250, sim.SpikeSourceArray,
                                    {'spike_times': [1000]},
                                    label='spike_source')


# Neuron Parameters
cell_params_exc = {
    'tau_m': 20.0, 'cm': 1.0, 'v_rest': -65.0, 'v_reset': -65.0,
    'v_thresh': -50.0, 'tau_syn_E': 5.0, 'tau_syn_I': 15.0,
    'tau_refrac': 0.3, 'i_offset': 0}

cell_params_inh = {
    'tau_m': 20.0, 'cm': 1.0, 'v_rest': -65.0, 'v_reset': -65.0,
    'v_thresh': -50.0, 'tau_syn_E': 5.0, 'tau_syn_I': 5.0,
    'tau_refrac': 0.3, 'i_offset': 0}

# Neuronal populations
pop_exc = sim.Population(500, sim.IF_curr_exp(**cell_params_exc),
                         label='excitatory_pop')

pop_inh = sim.Population(125, sim.IF_curr_exp(**cell_params_inh),
                         label='inhibitory_pop')


# Generate random distributions from which to initialise parameters
rng = sim.NumpyRNG(seed=98766987, parallel_safe=True)

# Initialise membrane potentials uniformly between threshold and resting
pop_exc.set(v=sim.RandomDistribution('uniform',
                                     [cell_params_exc['v_reset'],
                                      cell_params_exc['v_thresh']],
                                      rng=rng))

# Distribution from which to allocate delays
delay_distribution = sim.RandomDistribution('uniform', [1, 10], rng=rng)

# Spike input projections
spike_source_projection = sim.Projection(spike_source_array, pop_exc,
    sim.FixedProbabilityConnector(p_connect=0.05),
    synapse_type=sim.StaticSynapse(weight=0.1, delay=delay_distribution),
    receptor_type='excitatory')

# Poisson source projections
poisson_projection_exc = sim.Projection(poisson_spike_source, pop_exc,
    sim.FixedProbabilityConnector(p_connect=0.2),
    synapse_type=sim.StaticSynapse(weight=0.06, delay=delay_distribution),
    receptor_type='excitatory')
poisson_projection_inh = sim.Projection(poisson_spike_source, pop_inh,
    sim.FixedProbabilityConnector(p_connect=0.2),
    synapse_type=sim.StaticSynapse(weight=0.03, delay=delay_distribution),
    receptor_type='excitatory')
```

```
# Recurrent projections
exc_exc_rec = sim.Projection(pop_exc, pop_exc,
    sim.FixedProbabilityConnector(p_connect=0.1),
    synapse_type=sim.StaticSynapse(weight=0.03, delay=delay_distribution),
    receptor_type='excitatory')
exc_exc_one_to_one_rec = sim.Projection(pop_exc, pop_exc,
    sim.OneToOneConnector(),
    synapse_type=sim.StaticSynapse(weight=0.03, delay=delay_distribution),
    receptor_type='excitatory')
inh_inh_rec = sim.Projection(pop_inh, pop_inh,
    sim.FixedProbabilityConnector(p_connect=0.1),
    synapse_type=sim.StaticSynapse(weight=0.03, delay=delay_distribution),
    receptor_type='inhibitory')

# Projections between neuronal populations
exc_to_inh = sim.Projection(pop_exc, pop_inh,
    sim.FixedProbabilityConnector(p_connect=0.2),
    synapse_type=sim.StaticSynapse(weight=0.06, delay=delay_distribution),
    receptor_type='excitatory')
inh_to_exc = sim.Projection(pop_inh, pop_exc,
    sim.FixedProbabilityConnector(p_connect=0.2),
    synapse_type=sim.StaticSynapse(weight=0.06, delay=delay_distribution),
    receptor_type='inhibitory')


# Specify output recording
pop_exc.record('all')
pop_inh.record('spikes')


# Run simulation
sim.run(simtime=5000)


# Extract results data
exc_data = pop_exc.get_data('spikes')
inh_data = pop_inh.get_data('spikes')


# Exit simulation
sim.end()
```

Listing 4.2. An example for PyNN commands.

DOI: 10.1561/9781680836530.ch5

Chapter 5

Applications — Doing Stuff on the Machine

By Petruț Bogdan, Robert James, Gabriel Fonseca Guerra,
Garibaldi Pineda García and Basabdatta Sen-Bhattacharya

> *The Terminator's an infiltration unit. Part man, part machine.*
> *Underneath, it's a hyperalloy combat chassis,*
> *microprocessor-controlled, fully armored.*
> *Very tough … But outside, it's living human tissue.*
>
> — The Terminator

The SpiNNaker machine is flexible in terms of the applications that it supports. In part, this flexibility is given by the comparative ease of use of the substrate, namely the ARM processors. A varied range of applications is also encouraged by the software stack maturity discussed in Chapter 4. Using these high-level collections of software, a variety of plasticity mechanisms have been implemented to support various learning applications.

The following sections will cover a wide range of topics. We begin by first presenting an art exhibit and SpiNNaker's place in it – we start light. Then, we present a suite of approaches to engineer SNNs for a variety of computer vision tasks. Progressing through this chapter we present a large-scale model of a cochlea (SpiNNak-Ear [120]). This application is only possible on SpiNNaker because of general-purpose nature of the CPUs and the software written to support such generic, graph-based applications. From sensing to decision making, we present a model of a Basal Ganglia (BG), of course, simulated on SpiNNaker. Finally, we use SNNs as the method of solving constraint satisfaction problems.

5.1 Robot Art Project

We are a lab full of engineers. Art was as far away from our collective future projections for the platform as possible. So, once we were approached by Tove Kjellmark, a Swedish artist, with the idea for an exhibit involving humanoid robots and SpiNNaker, we immediately considered the issues and hurdles of such an attempt, not the least that of time and expectation management. The exhibition at the Manchester Art Gallery, named 'The Imitation Game' in honour of Alan Turing and his eponymous test, was to include several robotic pieces with the common theme of seeming intelligent in particular ways. The robotic entities present in the gallery would surely not pass Turing's test in any meaningful way, but that was not the plan anyway. To school children, laypeople and scientists alike, this was an artist's view at imitating life at the behavioural, albeit limited, level. At a basic level, these pieces would hint at the existence of something more than just Artificial Intelligence (AI). Tove Kjellmark would call it 'another nature', that is to say an elimination of the artificial boundaries between the technological, the mechanical and the natural. We would rather call it a conceptual step in a more important area of research, that of Artificial General Intelligence (AGI), as opposed to the narrow AI, nowadays present everywhere, the 'autistic savants' that tell you what objects you are looking at, what movies to watch next and what music to listen based on your listening habits.

Our involvement focused on the piece 'Talk' (pictured in Figure 5.1) that featured two robotic torsos sat cross-legged on comfortable chairs discussing a dream. They look at each other, gesture while talking, speak fluently and with appropriate cadence, sighs and pauses. If a human dares approach, they stop their conversation, turn their head to face the intruder to chastise them and wave them away.[1] Thus, SpiNNaker's task was to control the arms of the robots to perform realistic-looking arm movements in three regimes: idling, gesturing and silencing.

The focus of this undergraduate project was successful in revealing that SpiNNaker is capable of real-life, albeit impractical, applications. The individually packaged SpiNNaker boards would not be turned off for weeks at a time and would operate without flaw for over 7 hours a day for approximately 4 months in conjunction with the physical robots. As expected, maintenance visits to the Gallery would generally revolve around the robots or indeed the host computers, rather than any

[1] Robotic art gallery video presentation https://youtu.be/GaqgkyAIRBg

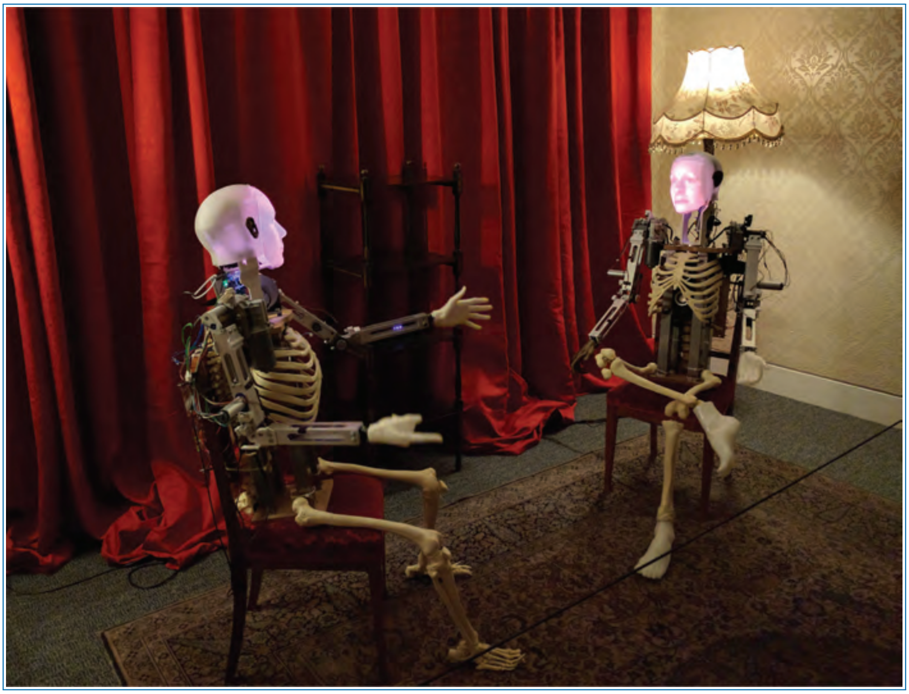


Figure 5.1. Display in 'The Imitation Game' exhibition at the Manchester Art Gallery, 2016, celebrating Manchester becoming European City of Science. Artist: Tove Kjellmark; School of Computer Science, Manchester: Petruț Bogdan, Prof. Steve Furber, Dr. Dave Lester, Michael Hopkins; Manchester Art Gallery Exhibitions Intern: Mathew Bancroft; Mechatronics Division, KTH, Stockholm: Joel Schröder, Jacob Johansson, Daniel Ohlsson, Elif Toy, Erik Bergdahl, Freddi Haataja, Anders Åström, Victor Karlsson, Sandra Aidanpää; Furhat Robotics: Gabriel Skantze, Jonas Beskow, Dr Per Johansson.

SpiNNaker intervention. It would seem that SpiNNaker would indeed be suited to neurorobotics applications [209], as discussed previously.

5.1.1 Building Brains with Nengo and Some Bits and Pieces

Two small PCs were used to control the two robots: the primary PC completely controls one of the robots and the arms of the other, while the secondary PC operates only the head of the other robot. The two distributed instances of the Furhat controller communicate through the network at key moments advancing the scripted dialogue. The primary PC is also responsible for communicating with the glorified distance sensor embodied in a Microsoft Kinect sensor, as well as the two stand-alone SpiNNaker boards. Both PCs control the actuators in the robotic arms using classical control theory; some translation is required between SpiNNaker's

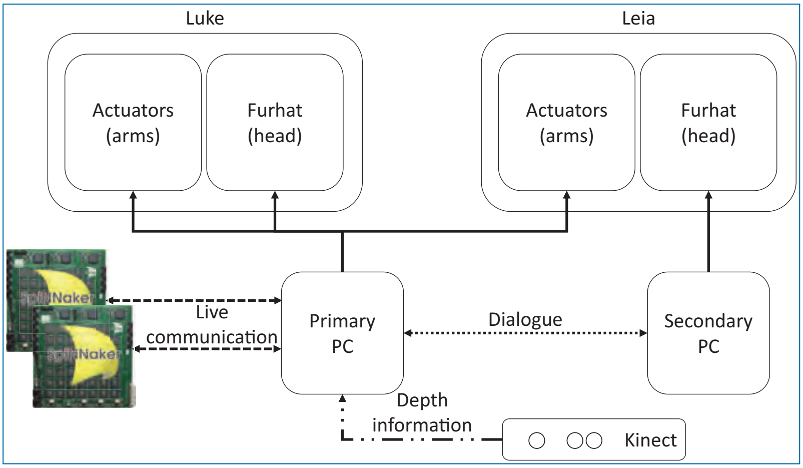


Figure 5.2. Hardware organisation diagram.

communication and these closed-loop control systems. Figure 5.2 reveals the flow of information involved in this project.

The previous chapter explained how SpiNNaker is usually controlled, using PyNN as a high-level network description language, viewing individual neurons as the main units of computation. Instead, here the Neural ENGineering Objects (Nengo) simulator bunches neurons together in ensembles (populations) and relies on their concerted activity to perform computation [53].

The way Nengo is built supports the implementation of a proportional-integral-derivative (PID) controller using a spiking neural substrate. A PID controller is a control loop feedback mechanism that continuously computes the error between the desired trajectory and the current position. The controller attempts to minimise the error as described by a weighted sum of a proportional, an integral and a derivative term. The proportional term accounts for moving towards the target at a rate dictated by the distance from it (cross track error). The derivative term considers the angle of the current trajectory compared to that of the desired trajectory (also called the cross track error rate), while the integral term is used to correct for accumulated errors that lead to a steady state error caused by, for example, external factors.

Consider the example of a driverless car positioned in a controlled environment with a trajectory precomputed for it to follow down the track in order to avoid some static obstacles. The goal is to try to follow the trajectory as closely as possible, so effects such as oscillations are not desired. In addition, the researchers at the facility have decided to see what would happen if at some point on the path they place a rock or pothole. They hope that the system would realise that it is drifting off course and apply a correcting turn. Figure 5.3 shows what this would look like in

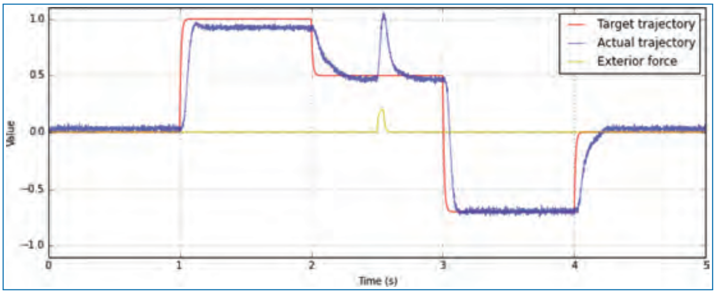
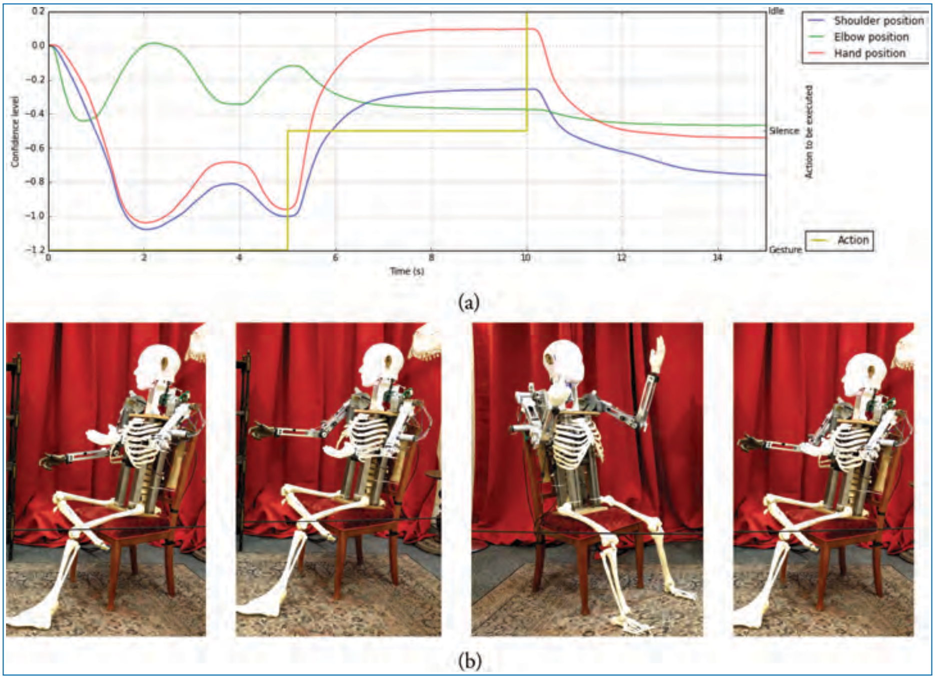


Figure 5.3. An example of trajectory following. In a real example, the trajectory would potentially not change so abruptly.



(a)

(b)

Figure 5.4. (a) A 15-second window of the operation of the control system running at the Manchester Art Gallery. This time period sees the robots going through all of the defined actions: gesture (the robot is talking), silence (the robot stopped talking to make a silencing gesture directed at an approaching visitor) and idle (the robot is not talking but listening to the other robot talk). (b) Robot poses corresponding to the Nengo simulation. The poses correspond to times 2, 4, 8 and 14.

Nengo. This is very similar to what can be done when controlling robot arm motors and servos.

Figure 5.4 shows the operation of one of the arms on a robot over a timespan of 15 seconds. During this time, the robot is issued three different commands in

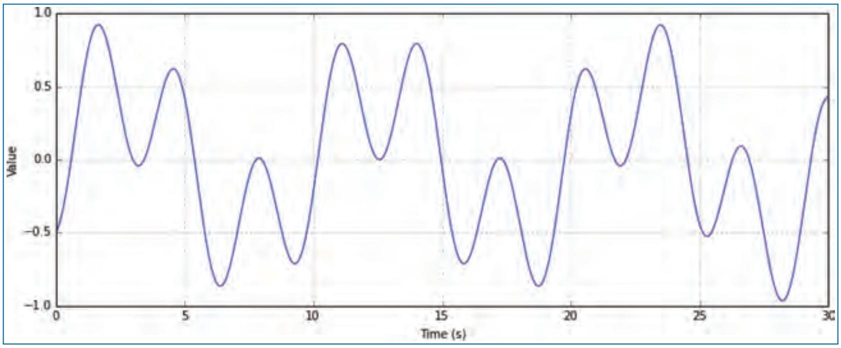


Figure 5.5. Gesturing movement of the robots computed as a function of time $f(t) = \frac{1}{2} * (sin(\frac{1}{1.6*t}) - cos(2 * t))$.

succession: gesture, silence and idle. While gesturing, the target position of each joint is given by a predetermined 'zero' or base position (hand-picked values that look natural in the physical exhibit) subtracted from a sinusoidal signal, namely the one in Figure 5.5. The incoming signal is transformed using a linear transformation for each joint individually to create a human like gesturing motion. Since the robots each has two arms, there is a dot product-based network inhibiting the arm that is not intended for use. Such arm selection is possible by creating a couple of predefined orthonormal vectors that represent the left and right directions. Based on the input direction vector for the system, a dot product is computed between it and the two previously mentioned bases so as to determine which direction is closest based on the angle. In the particular case where the vector is not significantly closer to any of the targets, the system accomplishes the desired action using both arms. The result of adding this level of control and inhibition is that the robot can now move one arm, or the other, or even both, thus allowing for more human mimetic behaviour.

When issued the action 'silence', the performing robot raises both lower arms into the air, in a defensive manner, signalled by external feedback from the head assembly, which turns to face the visitor and asks them to be silent. The action is achieved by inhibiting the neurons' spiking activity in the ensemble representing the 'zero' position and the 'sound' signal using the inhibiting output from an incorporated Basal Ganglia (BG) model. Analogously, idling is achieved by inhibiting the sound and silencing signals.

Because the exhibition took place in Manchester, no one else was around to maintain these robots, and we still had to experiment with realistic movement, we interacted with them for most of their stay at the Manchester Art Gallery. Most of these interactions took place during typical work hours, meaning that the gallery was usually populated by school children. It was surreal seeing the children interact with the robots. They weren't allowed to touch them of course, although that did not prevent them from trying. All of this assumes that they managed to enter the room: the usual first reaction to seeing them was fear. Once I had talked to the children's teachers and assured them that it was safe in the room, they would flock inside to witness the two humanoids in discussion. There was always someone watching from the doorway, too apprehensive to approach these mechanical beings, which were, essentially, only superficially intelligent. Nobody knew what they were talking about, but they were all fascinated with their 'silencing' phases as these provided the most audience interaction. These groups rarely stopped to read the plaque describing the exhibit, but surely this was a success in and of itself: SpiNNaker managed to work flawlessly for the entire duration of the exhibit; the same could not be said about the actuators and 3D-printed parts which had a much harder time.

5.2 Computer Vision with Spiking Neurons

Computational emulation of biological vision has been of interest for decades [249]. State-of-the-art computer vision systems use traditional image sensors for their inputs that differ greatly from those present in biology. In particular, ganglion cells in the mammalian retina emit signals when sufficient change in light intensity is sensed. Biology has successfully made use of event-based computation in vision (and other senses), and we should aim for the same in machine vision.

5.2.1 Feature Extraction

An important step in computer vision is to extract features from the input image. In traditional computer vision, this usually involves applying operations (e.g. convolutions and integrals) to the whole image, regardless of activity in the world, leading to high computational and bandwidth demands. Event-based computation diminishes these demands by processing only regions of the image that have changed.

Gabor-like Detection

To extract features, we can take inspiration from biological vision; Gabor-like filters are an example of a common abstraction which have an origin in biology and have been used in traditional computer vision [97]. These can be implemented using spiking neurons whose (immediate) receptive field is distance dependent and synapse weights are proportional to the ones computed by the Gabor function. Methods for transforming weight values have been proposed in the literature [185, 190] and in Chapter 7, we discuss a different approach.

Figure 5.6 (b–g) shows the result of filtering a Modified NIST (MNIST) digit (Figure 5.6(a)). The Gabor filters were generated using the following equations:

$$O(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \cos\left(2\pi \frac{x'}{\lambda} + \psi\right), \tag{5.1}$$

$$x' = x \cos\theta + y \sin\theta\,, \tag{5.2}$$

$$y' = -x \sin\theta + y \cos\theta\,, \tag{5.3}$$

where λ and ψ are the wavelength and phase of the sinusoidal component, respectively; θ is the orientation of the resulting stripes, σ is the standard deviation of the Gaussian component; and γ is the spatial aspect ratio. Parameters for the generation of Gabor kernels are presented in Table 5.1.

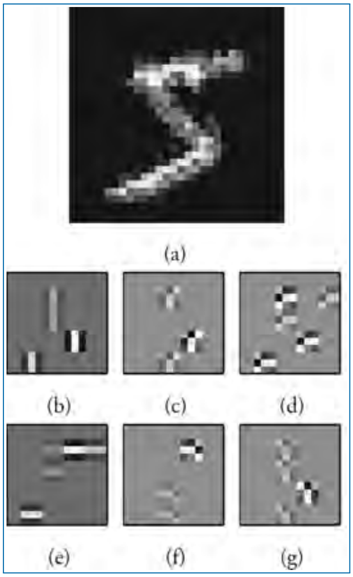


Figure 5.6. Results of Gabor-like feature extraction. (a) shows the input image converted to a spike train and later filtered using six Gabor kernels. (b–g) show the responses of each filtering population projected to the input space.

Table 5.1. Gabor filter parameters.

Width	Sampling	σ	λ	γ	ψ	θ
5	1	2	6	0.5	1.1	[0, 30, 60, 120, 150]

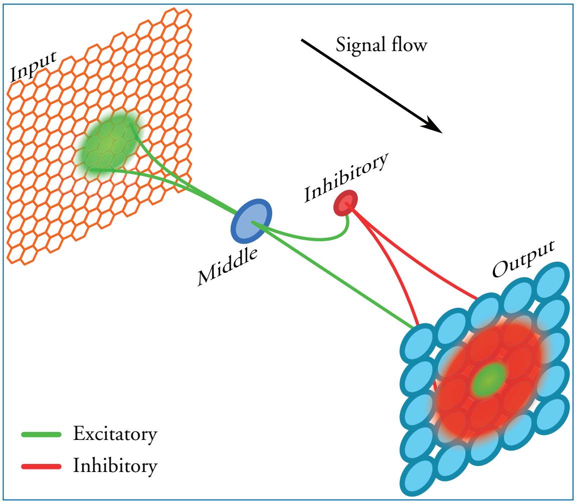


Figure 5.7. Connectivity motif for the blob-detecting network.

Blob Detector

Retinal connectivity has also been used as inspiration for key-point extraction [151]. A retina-inspired network can be used to convert visual input into a multi-scale representation from which blob-like features can be extracted [103]. In this three-layered network (Figure 5.7), the middle layer samples the input layer with receptive fields whose weights are computed using a Gaussian function. Different middle layer 'classes' sample the input with different parameters for their input kernels (i.e. width, σ). Each neuron in the middle layer drives a neuron in the output and, additionally, an inhibitory 'interneuron'. The purpose of the inhibitory neurons is to induce competition between the output layer neurons, reducing activity and pushing the output representation towards orthogonality. All neurons in the output layer compete to represent the input, and the extent to which the inhibitory neurons influence their neighbours is proportional to the cross-correlation of their input image kernels. This competition results in centre-surround receptive fields, as observed in biology.

As an example we took the same input image as in the Gabor filtering (Figure 5.6(a)), and its spike representation was processed by this blob-detection network using three different Gaussian kernel sizes. Figure 5.8 shows the output of the network; we can observe that the greatest activity is present in the mid-resolution class (Figure 5.8(b)) as it is a better fit to the input activity. The high-resolution class (Figure 5.8(a)) shows a behaviour similar to edge detection, typical of centre-surround filtering. Finally, as the receptive field for the low-resolution class is not a good fit for the input, there is little activity observed.

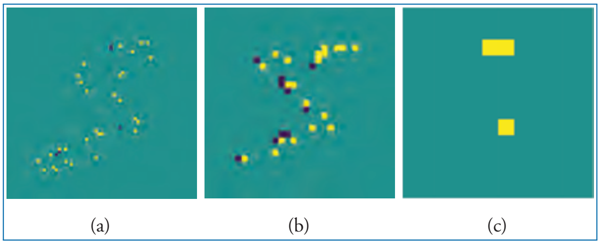


Figure 5.8. Results of blob-detection network. (a) High-, (b) middle- and (c) low-resolution neuron classes.

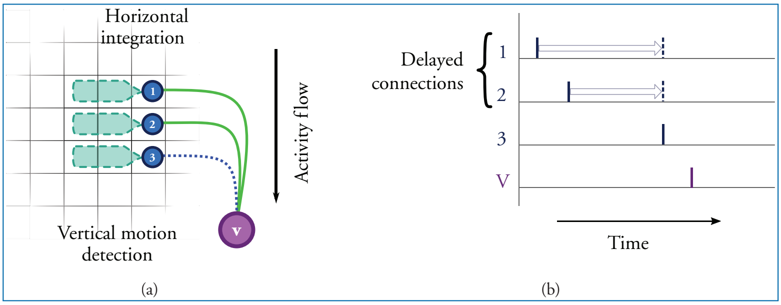


Figure 5.9. Motion sensing circuit. (a) Connectivity of the motion detection circuit using two different neurotransmitters (green-solid and blue-dashed). (b) Delayed lines allow spikes to reach the neuron body at the same time.

Motion Detection

Objects in the world are often moving, and since time is embedded in SNN simulations, we believe it is important to detect motion. A spiking version of a motion detector [103] was developed based on the connectivity of Starburst Amacrine Cells (SAC) [24, 58] and the Reichardt detector [24]. The motion detector network is illustrated in Figure 5.9(a); the principle of operation is composed of two factors: (i) delayed connections and (ii) the combination of two neurotransmitters. Delays are proportional to distance allowing incoming spikes triggered at different times and distances to arrive at (about) the same time (Figure 5.9(b)).

The two neurotransmitters allow activity from different regions of the input to be present at the correct time at the detector neuron (Figure 5.10(a) and 5.10(b)); one of the neurotransmitters decays at a slow rate, opening a window for the other transmitter (whose decay rate is high) to reach the detector.

We tested the circuit using a bouncing ball simulation; the ball moves in a 64 × 64 pixel window and when it bounces, it does so with a randomly selected speed in a range of 1 to 2 pixels. Figure 5.11 shows the outputs of easterly and westerly motion detection as red-dashed and green-solid lines, respectively. Ball motion is indicated by blue dots in the plot: the ball moved towards the north-east for

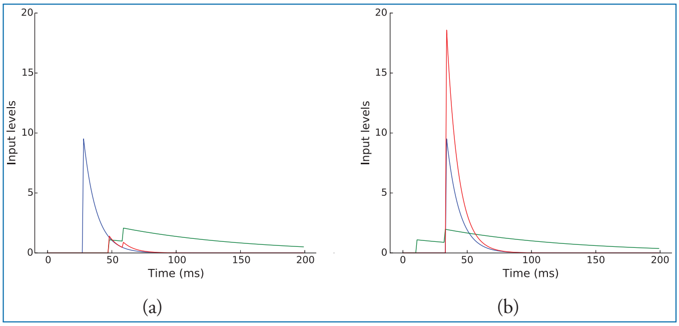


Figure 5.10. Interaction of transmitters in the motion sensing circuit. (a) When neurotransmitters (blue and green lines) do not reach the neuron within a temporal window, they will not induce sufficient current for the neuron to spike. (b) In contrast, when they reach the neuron in the right sequence, they will produce an activation.

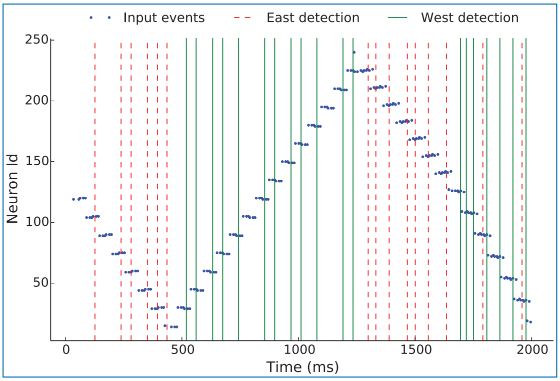


Figure 5.11. Output of the motion sensing circuit.

about 500 ms, then it bounced off a corner and moved in a south-westerly direction until ∼1250 ms; finally, it took off to the north-east again. In the first part (0 to ∼1250 ms) of the experiment, detection is near perfect although there are moments when the detectors fail to sense motion. In the last section (after ∼1250 ms), there are multiple false-positive detections which can be diminished by lateral competition of different directions. This circuit can detect apparent motion with an accuracy of 70%. A similar detector, though with learned connectivity, is described in Section 7.5.5.

5.3 SpiNNak-Ear — On-line Sound Processing

The SpiNNak-Ear system is a fully scaled biological model of the early mammalian auditory pathway: converting a sound stimulus into a spiking representation spread across a number of parallel auditory nerve fibres [119]. This system takes advantage of the generic digital processing elements on a SpiNNaker machine, enabling a Digital Signal Processing (DSP) application to be distributed across its massively parallel architecture. With the degree of parallel processing available for a SpiNNak-Ear implementation, one is able to generate a simulation of an ear to a biologically realistic scale (30,000 human cochlea auditory nerve fibres) in real time.

5.3.1 Motivation for a Neuromorphic Implementation

A conventional computer simulation can be carried out for large-scale auditory models – albeit with an inherent compromise in processing time due to serialised computation. However, an additional motivation for implementing a parallel simulation of the ear on SpiNNaker is the capability of handling a highly parallel interface between a model of the ear and the rest of the brain running on the same machine. SNNs that model later stages of the auditory pathway and cortical regions of the brain can be specified using the pre-existing SpiNNaker PyNN interface. Using a SpiNNak-Ear model that is already distributed across the same SpiNNaker network allows interfacing the auditory periphery with subsequent SNNs without incurring a data-flow bottleneck penalty.

5.3.2 The Early Auditory Pathway

The early auditory pathway, illustrated in Figure 5.12, begins with a sound pressure wave travelling into the outer ear and eventually displacing the Tympanic Membrane (TM) that separates the outer and middle ear. Inside the middle ear, the TM

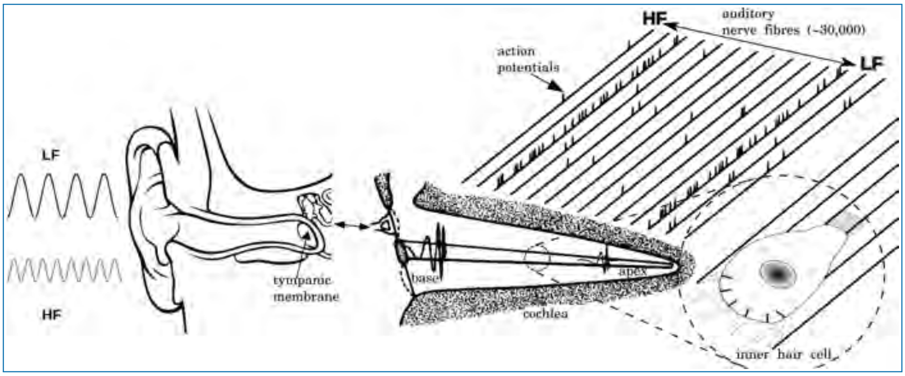


Figure 5.12. An uncoiled cochlea (right) with parallel auditory nerve fibres innervating single IHCs along the cochlea. The spiking activity due to two stimulus frequency components – High Frequency (HF) and Low Frequency (LF) – can be seen in the corresponding auditory nerve fibres.

connects to the cochlea via three ossicle bones to continue (and amplify) this displacement into the inner ear cochlea. The cochlea is a coiled, liquid-filled organ that converts the TM displacement into a series of travelling waves along its distance, from base to apex. The frequency components of the sound stimulus dictate the location along the cochlea that will experience the most displacement along its Basilar Membrane (BM). High frequencies are absorbed at the basal regions and progressively lower frequencies reach the apical regions of the cochlea. The cochlea is lined with many motion sensitive cells, known as Inner Hair Cells (IHCs), that detect the localised displacements of the BM. The IHCs act as the 'biological transducers' in the ear, converting physical sound-produced displacements into a corresponding spike code signal on the auditory nerve.

The modelling of every section of the cochlea's BM and the nearby IHCs can be described as being 'embarrassingly parallel', where the processing of each individual node (a Dual Resonance Non-Linear [DRNL] + IHC models) does not depend on any other neighbouring nodes. Therefore, we can model the processing of specific regions of the cochlea in a concurrent fashion.

5.3.3 Model Algorithm and Distribution

The algorithm used in SpiNNak-Ear is based on the MATLAB Auditory Periphery (MAP) model [159]. It separates the digital modelling of the ear into three separate modules representing ascending biological regions. The first module models the outer and middle ear (OME) using infinite impulse response filters. The second module mimics the sound stimulus frequency separation that occurs along the length of the cochlea using a filter bank of DRNL filters [150]. The final module represents the processing of the IHC and Auditory Nerve (AN) (IHC/AN) and is based on the algorithm described by Sumner *et al.* [245].

The complete SpiNNak-Ear module distribution is outlined in Figure 5.13; it consists of a single OME model instance and many DRNL and IHC/ANs instances depending on the number of cochlea frequency channels specified by the user. The data transfer between the OME model and connected DRNL models is performed using the SpiNNaker multicast-with-payload messaging method. This efficient routeing mechanism allows for the output of the OME model to be sent, a 32-bit sample at a time, as a multicast packet payload to all DRNL models located anywhere on the SpiNNaker machine. These incoming samples are stored in a local-to-core memory buffer and are batch processed when the designated processing segment size (96 samples) has been received. Following DRNL processing, the output 96 × 64-bit word segments are stored in a shared on-chip SDRAM memory circular buffer. This allows an efficient block data transfer between a 'parent' DRNL model and its 'child' IHC/AN models (always located on the same chip) necessary

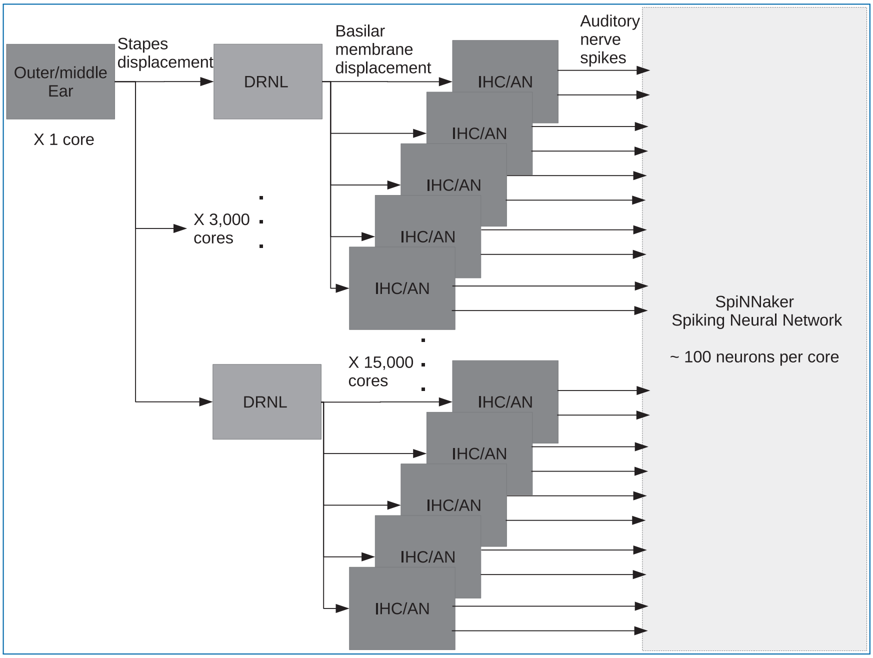


Figure 5.13. A schematic for the human full-scale early auditory path model distribution on SpiNNaker. The total number of cores for this simulation is 18,001 spanning across 1,500 SpiNNaker chips.

for real-time performance. The shared memory communication link that triggers a 'read from shared buffer' event in a child IHC/AN model is achieved using a multicast packet transmission from the parent DRNL model once it has processed a segment. Figure 5.14(a) illustrates these two data communication methods used in the full model system.

In the full system, the OME model application is triggered by the real-time input stimulus, after which the subsequent DRNL and IHC/AN models in the software pipeline are free to run asynchronously (event-driven) until the AN output stage. In a given simulation, to confirm that all model instances have initialised or have finished processing, we use 'core-ready' or 'simulation-complete' acknowledgement signals fed back through the network of all connected model instances to the parent OME model instance to ensure all cores are ready to process and data have been successfully recorded within the given time limits.

5.3.4 Results

The output from SpiNNak-Ear simulation is compared with conventional computer-based simulation results from the MAP model to ensure no significant

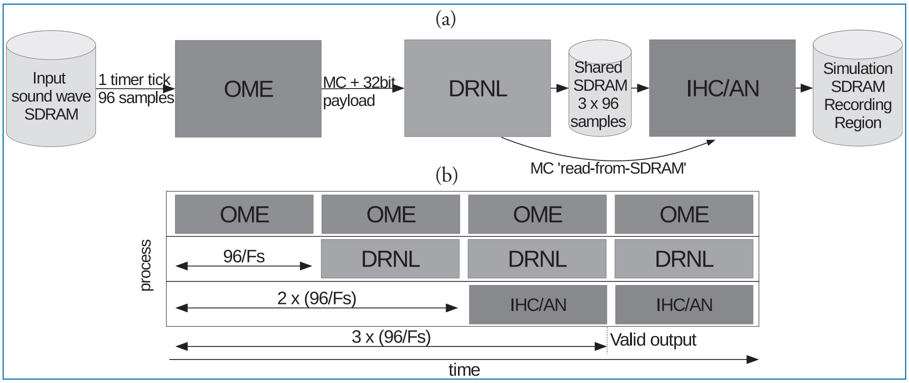


Figure 5.14. (a) The data passing method from input sound wave to the output of a single IHC/AN instance using MC and MC with payload message routeing schemes. (b) The pipeline processing structure used to achieve real-time performance.

numerical errors have occurred from computing the model algorithm on different simulation hardware. The outputs from both implementations are then compared with physiological experimental results to confirm the model's similarities to the biological processes it emulates.

In experimental neuroscience, the response from a stochastic auditory nerve fibre to an audio stimulus is measured over many repeated experiments and the subsequent recordings are often displayed in a Peri Stimulus Time Histogram (PSTH). The results, shown in Figure 5.15, show the time varying AN spike rates across 1 ms windows to a 6.9 kHz sinusoidal 68 dBSPL stimulus, first in Figure 5.15(a) from physiological data gathered by Westerman and Smith [265] and then from both model implementations in Figure 5.15(b). These results show both implementations produce a biologically similar response consisting of pre-stimulus firings of approximately 50 spikes/s, followed by a peak response at stimulus onset at around 800 spikes/s, decaying to an adapted rate in the region of 170 spikes/s. Finally at stimulus removal, rates significantly drop during an offset period before returning to spontaneous firing of approximately 50 spikes/s.

Figure 5.16 illustrates the energy consumed by MAP and SpiNNak-Ear implementations across the full range of model channels tested. Energy consumption has been calculated by multiplying the complete processing time by the total power rating of the hardware used (CPU at 84 W, single SpiNNaker chip at 1 W). Here we show that both implementations incur an increase in total energy consumed – but for different reasons. The MAP implementation running on a single, fixed power CPU uses more energy when the number of channels is increased due to the increase in serialised processing time. The neuromorphic hardware experiences an increase in energy consumed due to the increasing size of the machine used (number of

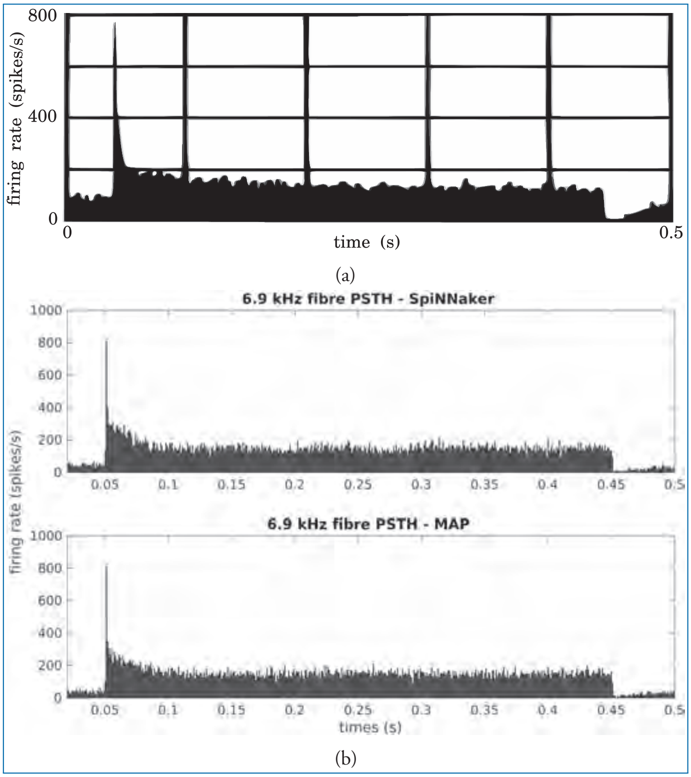


Figure 5.15. PSTH responses to 352 repetitions of a 400 ms 6.9 kHz 68 dBSPL stimulus from experimental data obtained by Westerman and Smith [265] of an HSR AN fibre in a gerbil (a) and the same experiment repeated for MAP and SpiNNaker implementations (b).

chips) with an increase in channels. The rate of increase in energy consumed due to number of channels on neuromorphic hardware is lower than the conventional serial CPU approach. This effect illustrates the basic philosophy that underlies the functionality of SpiNNaker (and biological) processing systems: complex computation on a modest energy budget, performed by dividing overall task workload across a parallel network of simple and power-efficient processing nodes.

5.3.5 Future Developments

A goal for the future of SpiNNak-Ear is to enable simulations with a live stream audio signal input. This has the potential to provide the user with an interactive visual representation of various regions of the brain to their current sound environment. Such a facility of ‘in-the-loop’ experimentation may assist in gaining further understanding of the important features of biological neural networks.

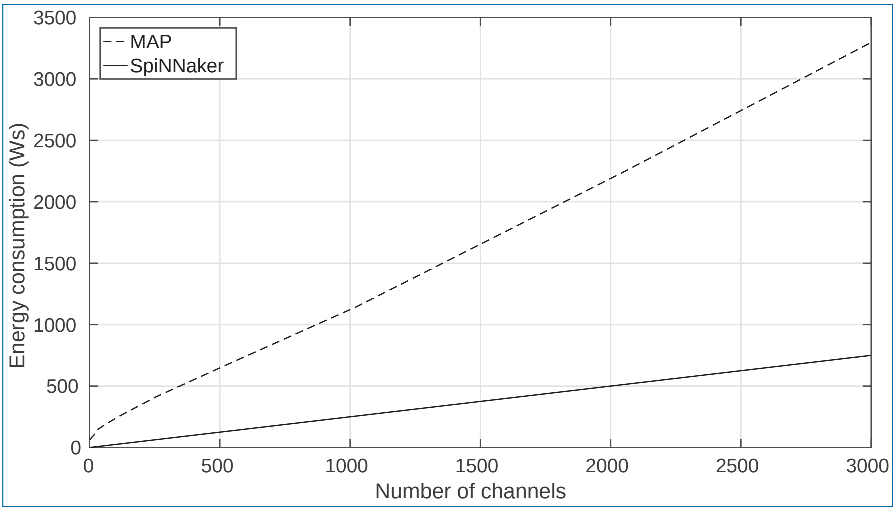


Figure 5.16. Average energy consumption from processing a 0.5 s sound sample from 2 to 3,000 channels on both MAP and SpiNNaker implementations. The MAP model is executed on a desktop computer (Intel Core™ i5-4590 CPU @ 3.3 GHz 22 nm technology) and SpiNNaker on a range of different sized SpiNNaker machines ranging from 1 to 1,500 chips (130 nm technology) scaled by the number of channels in a simulation.

The SpiNNak-Ear implementation on the SpiNNaker platform can be used in future investigation into the importance of the descending projections that feature between stages of the auditory pathway. It has been shown that descending projections may be providing useful feedback modulation to the incoming sound representation, 'tuning' the representations of learnt salient stimuli [252] and producing stimulus-specific adaptation in sensory neurons [153]. Therefore, if a research goal is to gain a full understanding of the auditory system, one must model it completely with multiple feedback projections. Implementing such connectivity across a large complex system in a computer simulation becomes an increasing burden on system communication resources. On the SpiNNaker hardware architecture, using the novel one-to-many multicast message routeing mechanism, additional descending projections can be integrated into simulations without incurring large overheads and with the ability to simulate real-time feedback to the user.

5.4 Basal Ganglia Circuit Abstraction

Here we present a biologically plausible and scalable model of the Basal Ganglia (BG) circuit, designed to run on the SpiNNaker machine [217]. It is based on the Gurney–Prescott–Redgrave model of the BG [84, 85]. The BG is a set of subcortical nuclei that are evolutionarily very old and appear in all vertebrates, enabling them to make decisions and take subsequent actions; obviously, therefore, computational modelling of the BG has been pursued by researchers with an interest in robotics [202]. The information on which the decision needs to be made, that is, the environmental circumstance, constitutes the input to the BG and is available via the thalamus and cortex. Output from the BG is the specific action that is decided upon, referred to as 'action-selection', and is relayed to the motor pathway for execution via the thalamus, cortex and other subcortical structures. The objective of our work on SpiNNaker is to build a 'basic building block' towards development of automated decision-making tools in real time.

A single neuro-computational unit in our BG model is simulated with a conductance-based Izhikevich neuron model. A columnar structure of the BG circuitry is shown in Figure 5.17; this forms the basic building block for our scalable framework and is thought to be a single 'channel' of action selection. The striatum forms the main input structure of the BG and receives excitatory glutamatergic synapses from both the cortex and the thalamus. The substantia nigra pars reticulata (SNr) forms the output structure of the BG and projects inhibitory efferents to the ventral thalamus and brainstem reticular formation.

The single-channel BG model is first parameterised on SpiNNaker to set the base firing rates for all model cell populations, informed by prior work by Humphries *et al.* [110]. Next, to simulate action selection by competing inputs, the model is

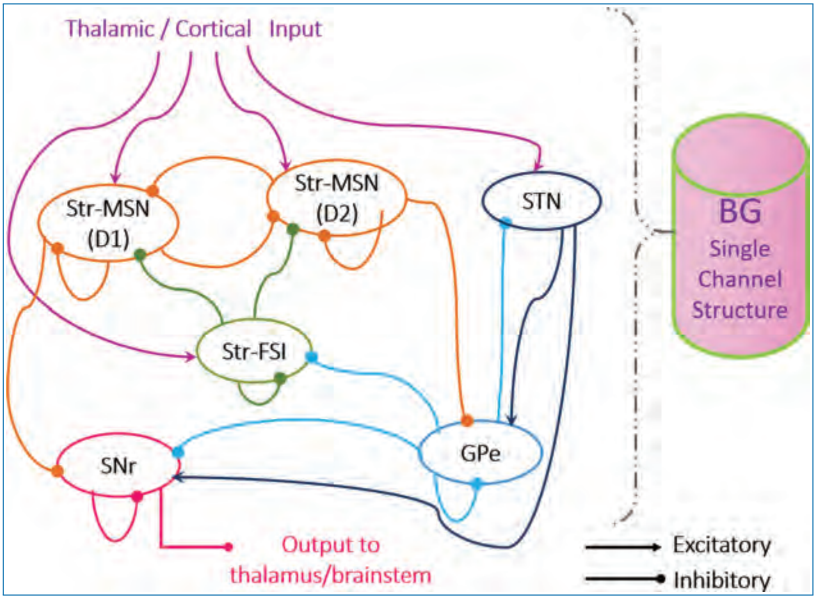

Thalamic / Cortical Input
Str-MSN (D1)
Str-MSN (D2)
STN
Str-FSI
SNr
GPe
Output to thalamus/brainstem
BG Single Channel Structure
Excitatory
Inhibitory

Figure 5.17. Single-channel action selection architecture.

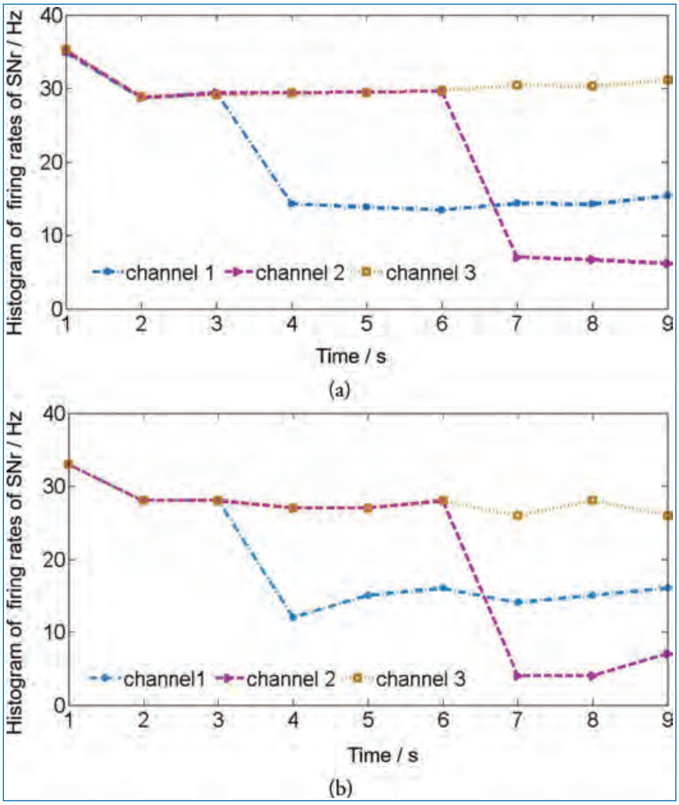


Figure 5.18. Demonstration of action selection in a 3-channel BG model on (a) SpiNNaker and (b) SpineML. All three channels have a 3 Hz Poisson input. At 3 seconds, a 15 Hz Poisson input is provided to channel 1 (blue), when the firing rate of the node drops, demonstrating disinhibition and therefore action selection by the node. At 6 seconds, channel 2 is provided with a 25 Hz input, and therefore, channel 2 now gets to select an action, as it is the overall winner with the lowest firing rate.

scaled up to three channels and tested with two competing inputs in the presence of a noisy background stimulus. Results are summarised in Figure 5.18(a). An input stimulus that is larger than the others is always the 'winner', indicated by a relative drop in the firing rate of the SNr population (representing the BG model output) in the competing channel. The reduced firing rate of the inhibitory SNr population implies a reduced inhibition of the thalamic/brainstem cells, which are the recipients of the BG output as mentioned above. This in turn means that the 'action' that is solicited by a relatively larger ('competing') input is now 'decided' by the BG circuit to be 'selected and acted upon', indicated by disinhibition of the target outputs. The model is tested with a competing input of 15 Hz in the presence of a noisy background input of 3 Hz. This is further confirmed by 'selection' of a larger input of 25 Hz provided in the presence of both 15 Hz and 3 Hz inputs. On both occasions, the largest input wins.

It is worth mentioning here that dopamine neurotransmitter-receptor levels are fundamental to facilitating decision-making and action selection by the BG. Here, the base parameters are tuned to simulate neutral dopamine levels; studying model dynamics with varying levels of dopamine will be carried out in future work.

To verify the model results simulated on SpiNNaker, the model is mapped to SpineML, an XML-based platform representing model attributes as 'components' and executing the models with SpineML_2_BRAHMS, a bespoke simulator that converts the SpineML model into machine code and runs it on a conventional computer. We aimed for the BG model implementation on SpineML to have the exact same network topology and neuron attributes as the SpiNNaker version and therefore retained all model connectivities and parameter values used in the latter. Model results on SpineML show qualitative similarity with those on SpiNNaker in terms of base firing rates of the single-channel BG model cell populations. Implementation of the three-channel model on SpineML, following the exact same implementation procedures as on SpiNNaker, demonstrates action selection by a larger input and is shown in Figure 5.18(b). Comparing Figures 5.18(a) and 5.18(b) shows an agreement between the functional and qualitative behaviour of the models simulated on SpiNNaker and SpineML. We believe that our comparative study will provide a basic framework for mapping SpiNNaker-based models to SpineML, as well as for performance benchmarking of SpiNNaker with conventional computers during neuronal simulation.

The single-channel BG model consists of 2.68×10^3 neurons and $\approx 0.68 \times 10^6$ synapses (estimated from projection probabilities). While each processor within a SpiNNaker chip is capable of simulating an upper limit of 256 neurons [217], memory requirements of the neuron model and synaptic connectivity for certain applications may cause this number to be reduced. In the current work, sPyNNaker maps the single-channel BG model onto 32 cores distributed across 2 SpiNNaker chips, residing on a single 48-chip SpiNNaker Board. Power consumption of the single-channel BG model executing on a 48-node board is measured using in-house Raspberry-Pi-based power measurement equipment [244]. Figure 5.19(a) shows that the single-channel model execution uses ≈800 mW. We have also observed that the model execution time is not affected by scaling up to three channels and is consistent at 100 s real time corresponding to a simulation time of 10 s. As power consumed during pre- and post-processing are negligible compared to that during model execution, we kept the post-processing time to a minimum; pre-processing times are handled by sPyNNaker and are not accessible to the user. Figure 5.19(b) shows a performance comparison between SpiNNaker and SpineML. The main constraints on SpiNNaker currently are the pre- and post-processing times that are

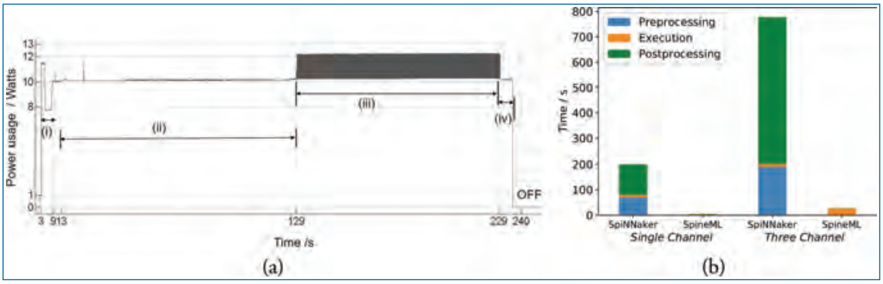


Figure 5.19. (a) The power consumption of the single-channel model using an in-house Raspberry-Pi-based measurement system connected to the SpiNNaker board [244]. The duration of recording power can be broken down into four regions: (i) booting the machine; (ii) pre-processing of data; (iii) model execution; (iv) post-processing (i.e. data extraction); the delay of around 4 s after booting the machine is inserted for clarity. The peak-to-peak power in region (iii) is 800 mW. (b) Performance analysis of single-channel and three-channel models running on both SpiNNaker and SpineML. Execution time on SpiNNaker, and pre- and post-processing times on SpineML are unaffected by scaling-up of model.

negligible on SpineML running on a 4-core 8 GB RAM desktop host machine, even for the scaled-up model. In contrast, execution time on SpiNNaker is not affected by model scaling; execution time for the SpineMLmodel is affected by scale. We believe that our comparative study will provide a basic framework for mapping SpiNNaker-based models to SpineML, as well as for performance benchmarking of SpiNNaker with conventional computers during neuronal simulation.

5.5 Constraint Satisfaction

When developing a biologically inspired hardware architecture, apart from looking for an improvement of our understanding of living matter, it is also desirable to explore the capabilities of the machine on the realm of more general problems in Mathematics and Computer Science. If the machine succeeds in representing or solving any of the well-defined abstract problems, or classes of problems, this means that it will be applicable to the specific cases that can be formulated under such formalisms, making the machine attractive for Physics and Engineering. The more general the class of problem, the broader the range of applications that will be covered and the better we will understand the capabilities of the design and, more importantly, we will understand its limitations. Understanding the limits of applicability of a computational approach is perhaps more important than evolving its capabilities. The reason being that some problems are indeed intractable in the sense that it does not matter how much we improve the speed, power consumption or size of our computers, there are families of problems which, despite being solvable in principle with infinite resources, will remain intractable at least until some exotic machine demonstrates an exponential speedup. Quantum and genetic computers, at least in theory, promise advances in this direction, but the practicalities currently seem to be out of scope. Worse than that are the undecidable or unsolvable problems. Hence, knowing the performance and complexity of a new computer architecture in the hierarchy of computable and incomputable problems will shed light on realistic directions for optimisation and improvement, avoiding the use of valuable time on aspects that will not add significant scientific or technological value.

Constraint Satisfaction Problems (CSPs) are a special family of problems that serve such a purpose. They are beautifully simple to formulate, yet they belong to the class of intractable problems (the NP-complete family). These are problems whose solutions are verifiable in Polynomial time (P), yet finding their solution requires supra-polynomial time as a function of the size of the problem. Actually, evidence suggests that the time complexity may be exponential, that is, a linear increase of the problem size results in an exponential increase of the required resources: time or space, memory or energy.

Formally, a CSP is defined by a set of variables $X = \{x_1, \ldots, x_N\}$ that take values over a set of discrete or continuous domains $D = \{D_1, \ldots, D_N\}$, such that a set of constraints $C = \{C_1, \ldots, C_m\}$ are satisfied. Each such constraint is defined as a tuple $C_i = \langle S_i, R_i \rangle$, where $R = \{R_1, \ldots, R_k\}$ are k relations over m subsets $S = \{S_1, \ldots, S_m : S_i \subseteq X\}$. In short, $CSP = \langle X, D, C \rangle$. Hence, the problem is defined over a combinatorial space whose size is on the order of $\overline{D}^N$, growing exponentially with N. Every solution to a CSP will have zero violations and include all variables in X. Hence, it will be represented by a global minimum of the cost hypersurface. If the problem has several solutions, the global minimum will be degenerate, one minimum existing for each solution. It is easy to see then that the difficulty of finding a solution for a CSP depends not only on the high dimensionality of its combinatorial space but also critically on the curvature of that space. Here, the curvature refers to how folded the space of possible evaluations of X is when measured against a scalar (energy or cost) function related to the number of unsatisfied constraints. If the cost function is strictly convex, there will be a single minimum and methods such as gradient descent will easily find it. Unfortunately, this is rarely the case.

With a geometrical representation of CSPs, it is easy to imagine solving the problem by travelling across the cost hypersurface, defined on some high-dimensional space, looking for a global minimum. Think of it as being like an adventurous explorer in the middle of the Amazon rain forest, perhaps searching for some previously uncontacted peoples. Having a helicopter would certainly be an advantage. Access to a satellite with high-resolution cameras and powerful zooming abilities will save you a lot of time. However, without any of them, you only have access to local information. Being inside the rain forest, any fancy equipment could only help you if it has access to some global parameter. That is the problem faced when solving CSPs. In general, you can provide the solver with local information, but any global information will only have low resolution, usually obtained from a discrete sampling over the state space, whose useful information content will decrease drastically with the curvature of the cost function.

Our aim in this section is to explore the representation and solution of CSPs on SpiNNaker. We stick to the use of spiking neurons, embracing the main purpose of the machine. However, bear in mind that SpiNNaker is a digital system, built from general-purpose processors, and has features that allow non-neural implementations. We will not consider those here. The following is extracted with minor modifications from Fonseca Guerra and Furber [61].

5.5.1 Defining the Problem

Consider a set $\mathcal{N}$ of neurons obeying a dynamic model defining the time evolution of a state variable, usually the membrane potential u_i and a threshold function θ_i that defines the generation of a spike event whenever the state variable reaches θ_i from below. After a spike u_i is forced below the threshold; it can, for example, be reset to a resting u_{rest} or reset u_{reset} membrane potential. The SNN is defined by the set $\mathcal{N}$ together with a set of synapses $\mathcal{S} \subseteq \mathcal{N} \mathrm{X} \mathcal{N}$ that define connections between pairs of neurons $\mathcal{N}_i$ and $\mathcal{N}_j$. Each synapse $\mathcal{S}_{i,j} \in \mathcal{S}$ has an associated weight parameter $w_{i,j}$ and a response function $\mathcal{R}_{i,j} : \mathbb{R}^+ \to \mathbb{R}$.

Let us also define the instantaneous state of the SNN as $\psi_t = \{n_1, n_2, \ldots, n_N\}$, that is, the ordered set of firing states $n_i \in \{0, 1\}$ for every neuron in $\mathcal{N}_i$ at time t. In SpiNNaker, ψ_t is well defined because time is discretised, generally in steps of 1 ms, and at each step, the firing state of a neuron is measurable. This definition avoids the need to track the membrane potential $u_i \in \mathbb{R}$ of each neuron at each particular time. Strictly speaking, u_i is represented by a 32-bit binary number in SpiNNaker so its resolution is finite.

In our implementation, each neuron $\mathcal{N}_i$ corresponds to a LIF neuron [238]. In this model, the dynamics of the membrane potential u are given by:

$$\tau_m \frac{du}{dt} = -u(t) + RI(t). \tag{5.4}$$

Here, τ_m is the membrane time constant, R is the membrane resistance and I is the external input current. Each time u reaches a threshold value θ a spike is elicited; such events are fully characterised by the firing times

$\{t^f \mid u(t^f) = \theta$ and $\frac{du}{dt} \lim_{t=t^f} > 0\}$. Immediately after a spike, the potential is reset to a value u_r, such that $\lim_{t \to t^{f+}} u(t) = u_r$. In our network, synapses are uniquely characterised by ω_{ij} and the inter-neural separation is introduced by means of a delay Δ_{ij}. In biological neurons, each spike event generates an electrochemical response on the post-synaptic neurons characterised by $\mathcal{R}_{i,j}$. We use the same function for every pair (i, j), and this is defined by the post-synaptic current:

$$j(t) = \frac{q}{\tau} e^{-\frac{t-t_0}{\tau}} \Theta(t - t_0), \tag{5.5}$$

where q is the total electric charge transferred through the synapse, τ is the characteristic decaying time of the exponential function, $t_0 = t^f + \Delta_{ij}$ is the arrival time of the spike and Θ represents the Heaviside step function. The choice of $\mathcal{R}_{i,j}$ potentially affects the network dynamics, and although there are more biologically realistic functions for the post-synaptic response, the use of the exponential function in Equation 5.5 constitutes one of our improvements over the previous studies on CSP through SNNs which used a simple square function.

In an SNN, each neuron is part of a large population. Thus, besides the background current $I(t)$, it receives input from the other neurons, as well as a stochastic stimulation from noisy neurons implementing a Poisson process. In this case, the temporal evolution of the membrane potential (Equation 5.4) generalises to:

$$\tau_m \frac{d}{dt} u = -u(t) + R \left[I(t) + \sum_j \omega_j \sum_f j(t - t_j^f) + \sum_k \Omega_k j(t - T_k) \right] \tag{5.6}$$

where the index f accounts for the spike times of principal neuron j in the SNN, Ω_k is the strength of the kth random spike, occurring at time T_k, and $j(.)$ is the response function of Equation 5.5. An SNN has the advantage that its microstate $\psi_t = \{n_1, n_2, \ldots, n_N\}$ at any time t can be defined by the binary firing state $n_i \in \{0, 1\}$ of each neuron $\mathcal{N}_i$, instead of the continuous membrane potentials $u_i \in \mathbb{R}$. Then, the set of firing times $\{t_i^f\}$ for every neuron $\mathcal{N}_i$, or equivalently the set of states $\{\psi_t\}$, corresponds to the trajectory (dynamics) of the network in the state space. The simulations in this work happen in discrete time (time step $=$ 1 ms) so, in practice, ψ_t defines a discrete stochastic process (e.g. a random walk). If the next network state $\psi_{t_{i+1}}$ depends on ψ_{t_i} but is conditionally independent of any ψ_{t_j} with $j < i$, the set $\{\psi_t\}$ also corresponds to a Markov chain. Habenschuss *et al.* [89] demonstrated that this is the case when using rectangular Post-Synaptic Potentials (PSPs) and a generalised definition of the network state, the validity of the Markov property for general SNNs could still depend on the dynamical regime and be affected by the presence of a non-zero probability current for the stationary distribution [39]. Each possible configuration of the system, a microstate ψ_i, happens with certain probability p_i and, in general, it is possible to characterise the macroscopic state of the network with the Shannon entropy (in units of *bits*) [221]:

$$S = -\sum_i p_i \log_2 p_i \tag{5.7}$$

and the network activity:

$$A(t) = \frac{1}{N} \sum_j^N \sum_f \delta(t - t_j^f) \tag{5.8}$$

To compute p_i and hence Equation 5.7, we binned the spikes from each simulation with time windows of 200 ms. In this type of high-dimensional dynamical system, sometimes the particular behaviour of a single unit is not as relevant as the collective behaviour of the network, described, for example, by Equations 5.7 and 5.8.

A constraint satisfaction problem $\langle X, D, C\rangle$ can now be expressed as an SNN as shown in the pseudo-code of Listing 5.1. We can do it in three basic steps: (a) create SNNs for each domain d_i of each variable, every neuron is then excited by its associated noise source, providing the necessary energy to begin exploration of the states $\{\psi\}$; (b) create lateral-inhibition circuits between all domains that belong to the same variable; (c) create lateral-inhibition circuits between equivalent domains of all variables appearing in a negative constraint and lateral-excitation circuits for domains in a positive constraint. With these steps, the resulting network will be a dynamical system representation of the original CSP. Different strategies can now be implemented to enforce the random process over states ψ_t to find the configuration ψ_0 that satisfies all the constraints. The easiest and proposed way of implementing such strategies is through the functional dependence of the noise intensity on time. The size of each domain population should be large enough to average out the stochastic spike activity. Otherwise, the system will not be stable and will not represent quasi-equilibrium states. As will be shown, it is the size of the domain populations what allows the system to converge into a stable solution.

The ensemble of populations assigned to every CSP variable x_i works as a Winner-Takes-All (WTA) circuit through inhibitory synapses between domain populations, which tends to allow a single population to be active. However, the last restriction should not be over-imposed, because it could generate saturation and our network will be trapped in a local minimum. Instead, the network should constantly explore configurations in an unstable fashion, converging to equilibrium only when satisfiability is found. The random connections between populations, together with the noisy excitatory populations and the network topology, provide the necessary stochasticity that allows the system to search for satisfiable states. However, this same behaviour traps some of the energy inside the network. For some problems, a dissipation population could be created to balance the input and output of energy or to control the entropy level during the stochastic search. In general, there may be situations in which the input noise acquired through stimulation can stay permanently in the SNN. Thus, the inclusion of more excitatory stimuli will saturate the dynamics at very high firing rates, which potentially could reach the limits of the SpiNNaker communication fabric. In these cases, inhibitory noise is essential too and allows us to include arbitrarily many stimulation pulses.

We demonstrate in the next section that the simple approach of controlling the dynamics with the stimulation intensities and times of the Poisson sources provides an efficient strategy for a stochastic search for solutions to the studied CSPs.

```
# define the CSP = <X,D,C> through a set of lists.
    X=list(variables)
    D=list(domains)
    S=list(subsets_of(X))
    R=list(relations_over(s_i in S))
    C=list(constraints = tuple(s_i,r_i))
#a) create an SNN for each variable with sub-populations for each domain.
    n = size_of_ensemble
    for variable x_i in X:
        for domain d_i in D:
            population[x_i][d_i] = create an SNN with n neurons
            noise_exc[x_i][d_i] = create a set of noise
            stimulation populations.
            apply_stimuli(noise[x_i][d_i], population[x_i][d_i])
            noise_inh[x_i][d_i] = create a set of noise
            dissipation populations.
            apply_dissipation(noise_inh[x_i][d_i], population[x_i][d_i])
#b) use inhibitory synapses to activate, on average, a single domain per variable
        for domain d_i in D:
            for domain d_j in D
                inhibition(population[x_i][d_i],population[x_i][d_j])
#c) map each constraint to an inhibitory or excitatory synapse.
    for constraint c_i in C:
        read subset s_i and relation r_i from c_i
        for variables x_i and x_j in s_i:
            for domain d_i in D:
                if constraint relation r_i <0:
                    inhibition(population[x_i][d_i], population[x_j][d_i])
                elif constraint relation r_i >0:
                    excitation(population[x_i][d_i], population[x_j][d_i])
```

Listing 5.1. Translation of a CSP into an SNN.

5.5.2 Results

In order to demonstrate the implementation of the SNN solver, we present solutions to some instances of Non-deterministic Polynomial time (NP) problems. Among the NP-complete problems, we have chosen to showcase instances of graph colouring, Latin squares and Ising spin glasses. Our aim is to offer a tool for the development of stochastic search algorithms in large SNNs. We are interested in

CSPs to gain understanding of the dynamics of SNNs under constraints, how they choose a particular state and their computational abilities. Ultimately, SNNs embedded in neuromorphic hardware are intended for the development of new technologies such as robotics and neuroprosthetics, constantly interacting with both external devices and the environment. In such applications, the network needs to adapt itself to time-varying constraints taking one or multiple decisions accordingly, making advances in stochastic search with SNNs a fundamental requirement for neuromorphics.

5.5.3 Graph Colouring

Consider a graph G defined by the ordered pair $\{V, E\}$, with V a set of vertices and E the set of edges connecting them. The graph colouring problem consists of finding an assignments of k colours to the elements of the graph (either V, E or both) such that certain conditions are satisfied [41]. In vertex colouring, for example, the colours are assigned to the elements of V in such a way that no two adjacent nodes (those connected by an edge) have the same colour. A particularly useful applications of this problem is the process of register allocation in compiler optimisation which is isomorphic to graph colouring [35]. Regarding time complexity, general graph colouring is NP-complete for $k > 2$. In the case of planar graphs, 3-colouring is NP-complete and, thanks to the four-colour theorem proved by Appel and Haken [5], 4-colouring is in P.

A division of a plane into several regions can be represented by a planar graph, familiar versions of which are the geographic maps. In Figure 5.20(a), we show the SNN-solver result of a satisfying 4-colouring of the map of the world where colours are assigned to countries such that no bordering countries have the same colour. We have used the list of countries and borders from the United Nations available in Mathematica Wolfram [113]. The corresponding connectivity graph of the world map in Figure 5.20(a) is shown in Figure 5.20(b). The insets in Figure 5.20(a) show the results of our solver for 3-colouring of the maps of the territories of Australia (bottom-right) and of Canada (top-left). Figure 5.20(c) and (d) show the time dependence of the entropy (top), firing rate (middle) and number of visited states (bottom) for the map of the world and of Australia, respectively. The colour code we use in these and the following figures is as follows: red means that the state in the current time bin is different from the one just visited, green represents the network staying in the same state and blue means that all constraints are satisfied. The dashed vertical lines mark the times at which noise stimulating (blue) or depressing (red) populations began to be active. The normalised spiking activity of the four colour populations for four randomly selected countries of the world map is shown in Figure 5.20(e) evidencing the competing behaviour along the stochastic

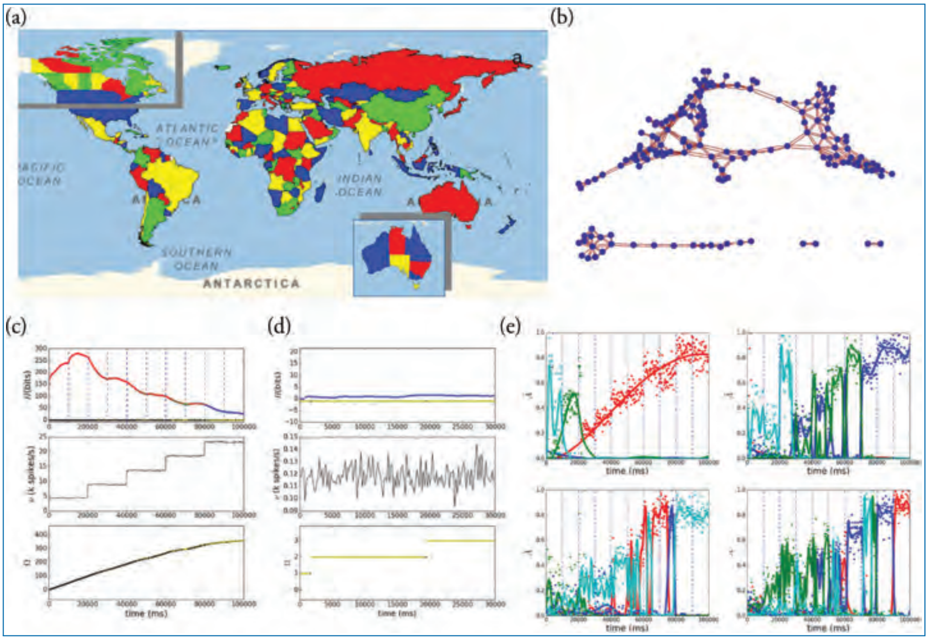
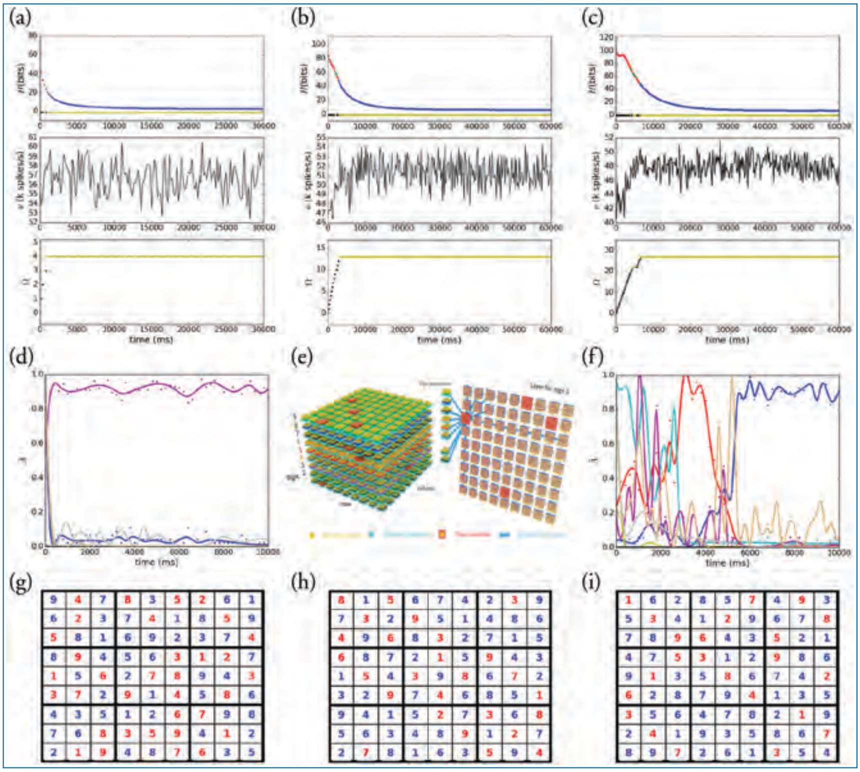


Figure 5.20. (a) Solution to the map colouring problem of the world with 4 colours and of Australia and Canada with 3 colours (insets). Figure (b) shows the graph of bordering countries from (a). The plots of the entropy H (top), mean firing spike rate ν (middle) and states count Ω (bottom) versus simulation time are shown in (c) and (d) for the world and Australia maps, evidencing the convergence of the network to satisfying stationary distributions. In the entropy curve, red codes for changes of state between successive time bins, green for no change and blue for the network satisfying the CSP. In the states count line, black dots mean exploration of new states; the dots are yellow if the network returns to states visited before. In (e), we have plotted the population activity for four randomly chosen CSP variables from (a), each line represents a colour domain.

search. Interestingly, although the network has converged to satisfaction during the last 20 s (blue region in Figure 5.20(c)), the bottom right plot in Figure 5.20(e) reveals that due to the last stimulation the network has swapped states preserving satisfaction, evidencing the stability of the convergence. Furthermore, it is noticeable in Figure 5.20(d) that new states are visited after convergence to satisfiability; this is due to the fact that, when multiple solutions exist, all satisfying configurations have the same probability of happening. Although we choose planar graphs here, the SNN can implement any general graph; hence, more complicated P and NP examples could be explored.

5.5.4 Latin Squares

A Latin square is defined as an array of $n \times n$ cells in which n groups of n different symbols are distributed in such a way that each digit appears only once in each row or column. The NP-completeness of solving a partially filled Latin square was demonstrated by Colbourn [38], and among the useful applications of such a problem, one can list authentication, error detection and error correction in coding theory. Here we choose the Sudoku puzzle as an instance of a Latin square, in this case, $n = 9$ and in addition to the column and row constraints of Latin squares, Sudoku requires the uniqueness of the digits in each 3×3 sub-grid. We show in Figure 5.21 the solution to an easy puzzle [57], to a hard Sudoku [89] and to the AI



(g)

9	4	7	8	3	5	2	6	1
6	2	3	7	4	1	8	5	9
5	8	1	6	9	2	3	7	4
8	9	4	5	6	3	1	2	7
1	5	6	2	7	8	9	4	3
3	7	2	9	1	4	5	8	6
4	3	5	1	2	6	7	9	8
7	6	8	3	5	9	4	1	2
2	1	9	4	8	7	6	3	5

(h)

8	1	5	6	7	4	2	3	9
7	3	2	9	5	1	4	8	6
4	9	6	8	3	2	7	1	5
6	8	7	2	1	5	9	4	3
1	5	4	3	9	8	6	7	2
3	2	9	7	4	6	8	5	1
9	4	1	5	2	7	3	6	8
5	6	3	4	8	9	1	2	7
2	7	8	1	6	3	5	9	4

(i)

1	6	2	8	5	7	4	9	3
5	3	4	1	2	9	6	7	8
7	8	9	6	4	3	5	2	1
4	7	5	3	1	2	9	8	6
9	1	3	5	8	6	7	4	2
6	2	8	7	9	4	1	3	5
3	5	6	4	7	8	2	1	9
2	4	1	9	3	5	8	6	7
8	9	7	2	6	1	3	5	4

Figure 5.21. SNN solution to Sudoku puzzles. (a–c) show the temporal dependence of the network entropy *H*, firing rate ν and states count Ω for the easy (g), hard (h) and AI escargot (i) puzzles. The colour code is the same as that of Figure 5.20. In (g–i), red is used for clues and blue is used for digits found by the solver. Figures (d) and (f) illustrate the activity for a random selected cell from (a) and from (c), respectively, evidencing competition between the digits, the lines correspond to a smoothing spline fit. (e) Schematic representation of the network architecture for the puzzle in (a).

Escargot puzzle, which has been claimed to be the hardest Sudoku. The temporal dependence of the network entropy H, firing rate ν and states count $\mathbf{\Omega}$ is shown in Figures 5.21(a)–(c), respectively, for the easy (5.21(g)), hard (5.21(h)) and AI escargot (5.21(i)) puzzles. In Figure 5.21(e), we show a schematic representation of the dimensionality of the network for the easy puzzle (g); each sphere represents a single neuron and synaptic connections have been omitted for clarity; the layer for digit 5 is represented also showing the inhibitory effect of a single cell in position (1,3) over its row, column, subgrid and other digits in the cell. In this case, the total number of neurons is $\approx 37\,k$ and they form $\approx 86\,M$ synapses.

One major improvement of our implementation with respect to the work of Habenschuss *et al.* [89] is the convergence to a stable solution; this is arguably due to the use of subpopulations instead of single neurons to represent the domains of the CSP variables as these populations were required to provide stability to the network. The use of the more realistic exponential post-synaptic potentials instead of the rectangular ones used by Habenschuss *et al.* [89] helps deliver a good search performance as shown in the bottom plots in Figure 5.21(a)–(c), where the solution is found after visiting only 3, 12 and 26 different states and requiring 0.8 s, 2.8 s and 6.6 s, respectively, relating well also with the puzzle hardness. It is important to highlight that the measurement of the difficulty level of a Sudoku puzzle is still ambiguous and our solver could need more complex strategies for different puzzles, for example, in the transient chaos-based rating the 'platinum blonde' Sudoku is rated as one of the hardest to solve, and although we have been able to find a solution for it, it is not stable, which means one should control the noisy network dynamics in order to survive the long escape rate of the model presented by Ercsey-Ravasz and Toroczkai [57]. We show in Figure 5.21(d) and (f) the competing activity of individual digit populations of a randomly chosen cell in both the easy and the AI escargot puzzles. The dynamic behaviour resembles that of the dynamic solver in Figure 2 of the work by Ercsey-Ravasz and Toroczkai [57] for this same easy puzzle and platinum blonde. Further analysis would bring insights into the chaotic dynamics of SNNs when facing constraints.

5.5.5 Ising Spin Systems

For each atom that constitutes a solid, it is possible to define a net spin magnetic moment $\vec{\mu}$ resulting from the intrinsic spin of the subatomic particles and the orbital motion of electrons around their atomic nucleus. Such magnetic moments interact in complex ways giving rise to a range of microscopic and macroscopic phenomena. A simple description of such interactions is given by the Ising model, where each $\vec{\mu}$ in a crystal is represented by a spin $\vec{S}$ taking values from $\{+1, -1\}$ on a regular discrete grid of points $\{i, j, k\}$. Furthermore, the interaction of the spins $\{\vec{S}_i\}$ is considered only between nearest neighbours and represented by a constant $J_{i,j}$ which determines if the two neighbouring spins will tend to align parallel $J_{i,j} > 0$ or anti-parallel $J_{i,j} < 0$ with each other. Given a particular configuration of spin orientations ω, the energy of the system is then given by the Hamiltonian operator:

$$\hat{\mathcal{H}} = -\sum_{i,j} J_{i,j}\vec{S}_i\vec{S}_j - h\sum_i S_i \tag{5.9}$$

where h is an external magnetic field that tends to align the spins in a preferential orientation [9]. In this form, each $J_{i,j}$ defines a constraint $C_{i,j}$ between the values $D = \{+1, -1\}$ taken by the variables $\vec{S}_i$ and $\vec{S}_j$. It is easy to see that the more constraints are satisfied, the lower the value of $\hat{\mathcal{H}}$ becomes in Equation 5.9. This simple model allows the study of phase transitions between disordered configurations at high temperature and ordered ones at low temperature. For ferromagnetic $J_{i,j} > 0$ and antiferromagnetic $J_{i,j} < 0$ interactions the configurations are similar to those in Figure 5.22(d) and (e) for 3D lattices. These correspond to the stable states of our SNN solver when the Ising models for $J_{i,j} > 0$ and $J_{i,j} < 0$ are mapped to an SNN using Algorithm 5.1 and a 3D grid of 1,000 spins. Figure 5.22(g) shows the result for a 1D antiferromagnetic spin chain. It is interesting to note that the statistical mechanics of spin systems has been extensively used to understand the firing dynamics of SNNs, presenting a striking correspondence between their behaviour even in complex regimes. Our framework allows the inverse problem of mapping the SNN dynamics to spin interactions. This equivalence between dynamical systems and algorithms has largely been accepted and we see an advantage in computing directly between equivalent dynamical systems. However, it is clear that the network parameters should be adequately chosen in order to keep the computation valid.

If instead of fixing $J_{i,j}$ to some value U for all spin pairs $\{(i, j)\}$ one allows it to take random values from $\{U, -U\}$ with probabilities p_{AF} and p_{FM}, it will be found that certain interactions would be frustrated (unsatisfiable constraints). Figure 5.22(f) illustrates the frustration with three antiferromagnetic interacting spins in a way that any choice of orientation for the third spin will conflict with one or the other. This extension of the Ising model when the grid of interactions is a random mixture of AF and FM interactions was described by Surungan *et al.* [246]. The model is the representation of the spin glass systems found in nature; these are crystals with low concentrations of magnetic impurities that, due to the frustrated interactions, are quenched into a frozen random configuration when the temperature is lowered (at room or high temperature the magnetic moments of a material are constantly and randomly precessing around their average orientation).

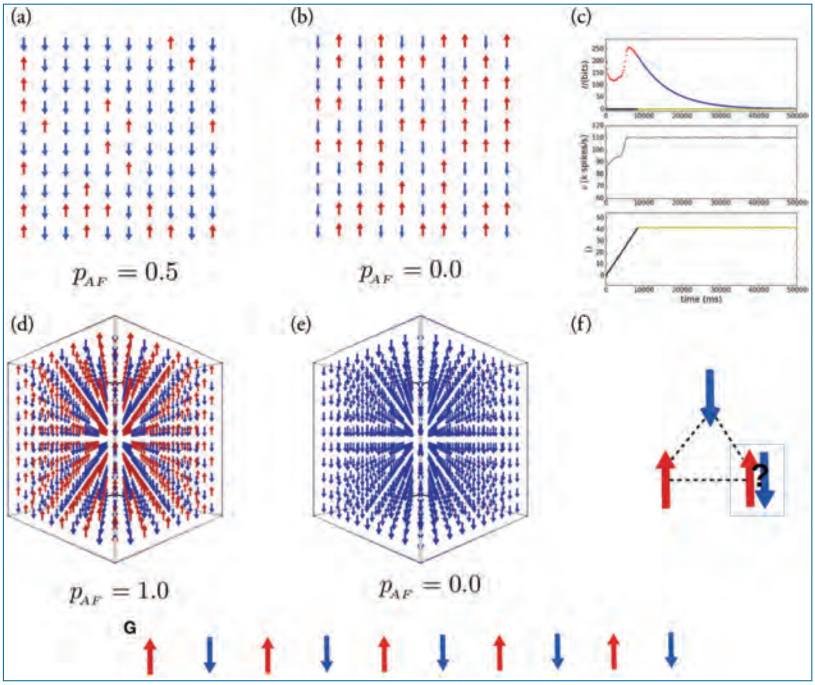


Figure 5.22. SNN simulation of Ising spin systems. (a) and (b) show two 2-dimensional spin glass quenched states obtained with interaction probabilities $p_{AF} = 0.5$ and $p_{AF} = 0.1$. The results for the three-dimensional lattices for CSPs of 1,000 spins with ferromagnetic and antiferromagnetic coupling constant are shown in (e) and (d), respectively. In (c) are plotted the temporal dependence of the network entropy *H*, firing rate ν and states count Ω during the stochastic search for the system in (d). (f) illustrates the origin of frustrated interactions in spin glasses. (g) depicts the result for the one-dimensional chain.

The statistical analysis of those systems was fundamental for the evolution of artificial neural networks and machine learning. Furthermore, the optimisation problem of finding the minimum energy configuration of a spin glass has been shown to be NP-complete [9]. The quenching of the grid happens when it gets trapped in a local minimum of the state space of all possible configurations. In Figure 5.22(a) and (b), we show a quenched state found by our SNN with $p_{AF} = 0.5$ and $p_{AF} = 0.1$, respectively. A spin glass in nature will often be trapped in local minima and will need specific temperature variations to approach a lower energy state; our SNNs replicate this behaviour and allow for the study of thermal processes, controlling the time variation and intensity of the excitatory and inhibitory stimulations. If the underlying stochastic process of such stimulations is a good representative of heat in solids, they will correspond to an increase and a decrease of temperature, respectively, allowing, for example, the implementation of simulated annealing optimisation. Figure 5.22(c) shows the time evolution of the entropy, firing rate and states count for the antiferromagnetic 3D lattice of Figure 5.22(d). Similar plots, but converging to unsatisfying states, are found for the spin glasses in Figure 5.22(a) and (b). In the case of the ferromagnetic lattice in 5.22(e) with a very low noise, the network immediately converges to a solution. If the noise is high, however, it is necessary to stimulate the network several times to have a perfect ordering. This is because more noise implies more energy to violate constraints; even in nature, magnetic ordering is lost at high temperatures.

DOI: 10.1561/9781680836530.ch6

Chapter 6

From Activations to Spikes

By Francesco Galluppi, Teresa Serrano Gotarredona, Qian Liu and Evangelos Stromatias

> *Tackling real-world tasks requires being comfortable with chance, trading off time with accuracy, and using approximations.*
>
> — Brian Christian in Algorithms to Live By: The Computer Science of Human Decisions

Deep learning has become the answer to an increasing number of AI problems since Hinton *et al.* [98] first proposed the training method of the Deep Belief Network (DBN). Machine learning is an extremely interesting space in 2019. The past few years have seen DeepMind create a narrow AI to master the game of Go and defeat Lee Sedol, a professional Go player [227]. Raising the ante, OpenAI designed a system to play a co-operative computer game and beat a professional team at it.[1] The game in question was DOTA 2 (Defense of the Ancients) a multiplayer online battle arena where each player (or AI) generally controls a single character. The same year, DeepMind showcased their system (AlphaStar) playing an arguably even more difficult game – Starcraft 2 – a real-time strategy game where each opponent can control up to 200 units, while also needing to focus on more abstract goals such as maintaining a functioning economy and production facilities [260]. These are just

[1] https://openai.com/five/

examples of applications of one AI technique (Reinforcement Learning) to a very narrow field, although even these have much wider applicability.

However, deep learning is not new 'magic', but rather has a history over a few decades. An overview of some popular Artificial Neural Networks (ANNs) is offered below. The rest of the chapter reveals how one can use the application-focused insights from Deep Neural Networks (DNNs) to engineer SNNs and an approach to convert pre-trained networks to use spikes.

6.1 Classical Models

We call the well-known and widely used deep learning models 'classical' and give a brief introduction to those models in this section. As mentioned above, the first break-through in training deep (>2 layer) networks was the greedy layer-wise strategy [98] proposed to train stacked Restricted Boltzmann Machines (RBMs). Shortly after, this method was proved also to be efficient for training other kinds of deep networks including stacked autoencoders (AEs) [13]. RBMs and AEs are suitable for dimensionality reduction and feature extraction when trained with unsupervised learning on unlabelled data. In 2012, using such an unsupervised deep learning architecture, the Google Brain team achieved a milestone in the deep learning era: the neural network learned to recognise cats by 'watching' 10 million images generated from random frames of YouTube videos [137].

Convolutional Neural Networks (ConvNets) are vaguely inspired from biology and the significant discovery of Hubel and Wiesel that simple cells have a preferential response to oriented bars (convolution) and complex cells collate responses from the simple ones (pooling); it is believed that these represent the basic functions in the primary visual cortex in cats [109]. These simple cells fire at a high frequency to their preferred orientation of visual stimuli within their receptive fields, small subregions of the visual field. Meanwhile, a complex cell corresponds to the existence of a pattern within a larger receptive field but loses the exact position of the pattern. The NeoCognitron [63] was the first network to mimic the functions of V1 simple and complex neurons in an ANN, and later, this feature detection of single cells was improved by sharing weights among receptive fields in LeNet-5 [138]; typically, ConvNets follow the same principle to this day. The mechanism of shared weights forms the essence of convolution in a ConvNet, which hugely reduces the number of trainable parameters in a network. The usual procedure to train ConvNets is a supervised one and is known as the back-propagation algorithm; it relies on the calculus chain rule to send error signals through the layers of the network starting from the output and ending at the input.

The most significant examples of ConvNet have dominated the best performances in the annual ImageNet Challenge [215]: AlexNet [132], VGG Net [228], GoogLeNet [249], ResNet [93] and MobileNet [108].

Despite the powerful capabilities of these feed-forward deep networks, sequence processing is a challenge for them since the size of the input and output vectors are constrained to the number of neurons. Thus, Recurrent Neural Networks (RNNs), containing feed-back connections, are ideal solutions for dealing with sequential information since their current output is always dependent on the previous 'memory'. As training mechanisms have become more mature, for example, using Long Short-Term Memory (LSTM) [99], RNNs have shown great success in many natural language processing tasks: language modelling [166], machine translation [247], speech recognition [83] and image caption generation [125].

The current trend in deep learning is to combine Machine Learning (ML) algorithms towards more complex objectives such as sequential decision-making and data generation.

Reinforcement Learning (RL) is inspired from animal behaviour when agents learn to make sequential optimised decisions to control an environment [248]. To address complex decision-making problems in practical life, RL requires a sufficiently abstract representation of the high-dimensional environment. Fortunately, deep learning nicely complements this requirement and performs effectively at dimensionality reduction and feature extraction. Advances in RL techniques, such as asynchronous advantage actor-critic (A3C) [167], are what allowed DeepMind and OpenAI to perform the feats presented at the beginning of this chapter.

Generative Adversarial Networks (GANs) [80] are proposed for training generative models of complex data. Instead of training discrimination networks (e.g. image classification using ConvNets) and generation networks (e.g. data sampling on RBMs) separately with different objectives, GANs train two competing networks – one the discriminator, the other the generator – simultaneously by making them continuously play games with each other. Thus, the generator learns to produce more realistic data to fool the discriminator, while the discriminator learns to become better at distinguishing generated from real data. Exciting achievements have been reported in generating complex data such as realistic image generation based on descriptions in text [203].

6.2 Symbol Card Recognition System with Spiking ConvNets

The ConvNet is the most commonly used machine learning architecture for image recognition. It is a biologically inspired generic architecture for intelligent data

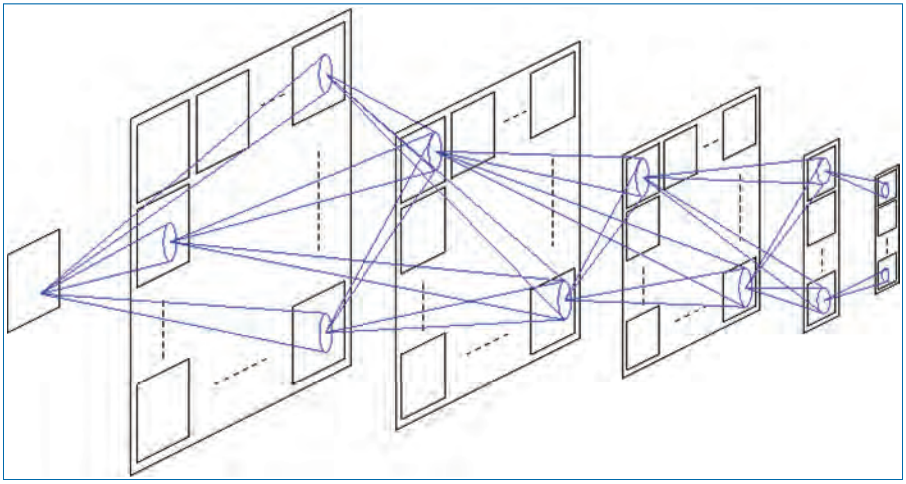


Figure 6.1. Generic ConvNet architecture.

processing [139]. The generic architecture of a ConvNet for visual object recognition is depicted in Figure 6.1. The visual scene coming out of the retina is fed to a sequence of layers that emulate the processing layers of the brain visual cortex. Each layer consists of the parallel application of 2D-filters to extract the main image characteristics. Each image representation obtained is named a feature map. The first layer extracts oriented edges of the image according to different orientations and different spatial scales. The subsequent layers combine the feature maps obtained in the previous layers to detect the presence of combinations of edges, detecting progressively more complex image characteristics, until achieving the recognition of complex objects in the higher levels. Along the ConvNet layers, the sizes of the feature maps are progressively reduced through applying image subsampling. This subsampling process is intended to introduce invariance to object size and position.

In conventional AI vision systems, the ConvNet architectures are used in a frame-based manner. A frame representing the particular scene to be analysed is fed to the architecture. The output of the different convolutional layers is computed in a sequential way (layer by layer) until a valid output is obtained in the upper layer indicating the category of the recognised object. However, this is not what happens in biological brains. In a biological system, the retina 'pixels' send, in an asynchronous way, sequences of spikes representing the visual scene. Those spikes are sent through the optic nerve to the visual cortex where they are processed as they arrive by the subsequent neuron layers with just the delay of the spike propagation and neuron processing.

We have used the SpiNNaker platform to implement a spiking ConvNet. Each time a spike is generated by a neuron in a layer, the spike is propagated to the connected neural populations of the next layer and the weights of the corresponding 2D-filter kernels are summed to the neuron states of the subsequent connected layer. That way the convolution is performed on the fly in a spike-based manner [220].

The input stimulus provided to the system is a flow of spikes representing the symbols of a poker card deck passing in front of a DVS [141, 143, 219] at high speed. We used an event-driven clustering algorithm [47] to track the passing symbols and, at the same time, we adjusted the tracking area to a 32 × 32 resolution. Each symbol passed in 10–20 ms producing 3 k–6 k spikes. The 40 symbols passed in 0.95 s generating a total of 174,643 spikes.

To achieve real-time recognition with, at the same time, reproducibility of the recordings, we loaded the spike sequence onto a data player board [218]. The data player board stores the neuron addresses and timestamps of the recorded spikes in a local memory and reproduces them as events through a parallel AER link in real time. The parallel AER events are converted to the 2-of-7 SpiNNaker protocol and fed in real time to the SpiNNaker machine.

The particular ConvNet architecture used for the card symbol recognition task is detailed in Figure 6.2. It consists of three convolutional layers (C1, C3 and C5) interleaved with two subsampling layers (S2 and S4) and a final fully-connected category layer (C6). Table 6.1 details the numbers and sizes of the feature maps as well as the sizes of the kernels in each layer.

The kernels in the first layer are a set of six Gabor filters in three different orientations and for two different spatial scales and are fixed, not trained. The rest of

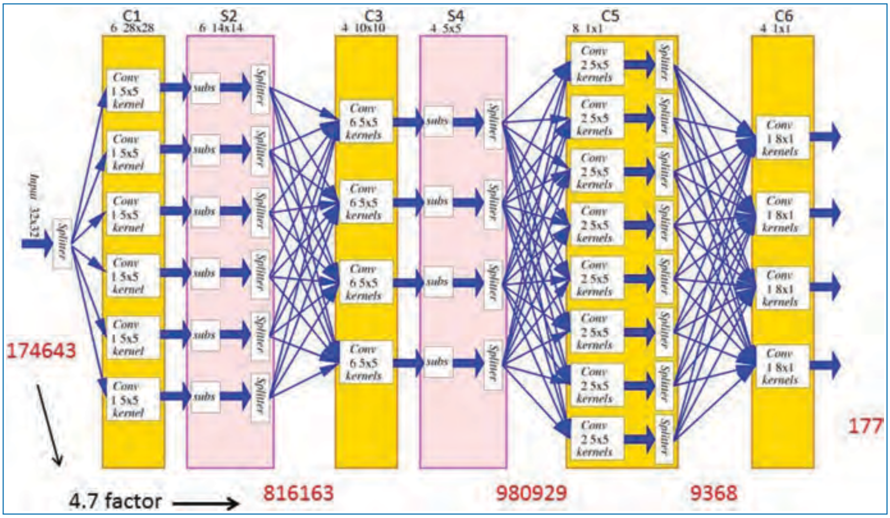


Figure 6.2. ConvNet architecture for card symbol recognition.

Table 6.1. Number and size of layers in card symbol ConvNet architecture.

	ConvNet structure					
	C1	**S2**	**C3**	**S4**	**C5**	**C6**
Feature maps (FM)	6	6	4	4	8	4
FM dimension	28 × 28	14 × 14	10 × 10	5 × 5	1 × 1	1 × 1
Kernel size	10 × 10	–	5 × 5	–	5 × 5	1 × 1
Number of kernels	6	–	24	–	32	32
Number of weights	600	–	600	–	800	32
Trainable weights	0	–	600	–	800	32

the network weights are trained using frames and a method to convert the weights to the spiking domain [190].

6.2.1 Spiking ConvNet on SpiNNaker

One of the peculiarities of a ConvNet architecture is the weight-sharing property. The weights of the kernels that connect two neurons in two different feature maps do not depend on the particular neurons but just on the relative positions of the two neurons in the origin and destination feature maps. Because of that 'weight-sharing' property, the number of synaptic weights that must be stored for a neuronal population is greatly reduced compared to populations with full connectivity and independent non-shared weights. To optimise the processing speed, the SpiNNaker tool flow was modified to admit a special 'convolution connector'. The 'convolution connector' is shared by all the neurons belonging to the same convolutional feature map population and contains the kernel weights which are stored in the local DTCM of the corresponding population. This solution avoids the reading of the kernel weights from the external SDRAM each time a spike arrives to the convolution module. Each time a spike arrives at a convolution module, depending on the source population of the incoming spike; the corresponding kernel is read from the DTCM memory and the neuron states of the neighbour pixels are updated correspondingly. If any of the updated neurons passes the firing threshold, an output spike is generated and immediately sent to the next processing layer, in the usual way neural systems are implemented on SpiNNaker. In this way, the ConvNet is truly spike- or event-driven.

Another characteristic of the ConvNet is that most of the neuron parameters (such as neuron voltage thresholds, voltage reset levels, leakage rates and refractory times) are shared by all the neurons in the same population. Only the particular neuron state and firing times are individual for each neuron. In the standard SpiNNaker tool chain, all the neuron parameters are replicated and stored individually for each neuron in the DTCM. Thus, the DTCM capacity sometimes limits the number of neurons that can be implemented per core. Here the tool chain was also modified to distinguish between the parameters that are individual for each neuron and the parameters shared by all the population. With this approximation, we are able to implement 2,048 convolution neurons per core, where this number is determined by the maximum number of addressable neurons supported by the routeing scheme.

6.2.2 Results

To test the recognition rate, we used a test sequence of 40 32 × 32 tracked symbols obtained from the events recorded with a DVS [219]. As already explained, the recording consists of a total of 174,643 spikes encoded as AER events.

We first tested the correct functionality of the ConvNet for card-symbol classification programmed on the SpiNNaker board at low speed. For this experiment, we multiplied by a factor 100 the timestamps of all of the events of the sequence that was reproduced by the data player board. To maintain the same classification capability as the ConvNet architecture optimised for card symbol recognition, we had to multiply the time parameters (the refractory and leakage times) of the network by the same factor 100. In Figure 6.3, we reproduce four snapshots of the composition grabbed with the jAER board of the input stimulus and the output category obtained with the SpiNNaker ConvNet classifier. As can be seen, correct classification of the four card symbols is obtained. These snapshots are generated by collecting and histogramming events with the jAER [178] board over 1.2 ms. The classification of the test sequence [190] of 40 card symbols slowed down a factor 100 was repeated 30 times. During the appearance time of each input symbol, the number of output events generated by the correct output category was counted as well as the number of output events generated by each of the other three output categories. The classification is considered successful if the number of output events of the correct category is the maximum. The mean of the success classification rate was 97% for the 30 repetitions of the experiment, with a maximum of 100% and a minimum of 93% in the success classification rate, thus achieving a recognition success rate slightly higher than the one obtained in the software real-time experiment [190].

Once we had tested that the SpiNNaker ConvNet classifier functionality was correct, we tested its maximum operation speed. For that purpose, we repeated the experiment for different slow-down factors of the event timings of the input stimulus sequence while, at the same time, we applied the same factor to the ConvNet

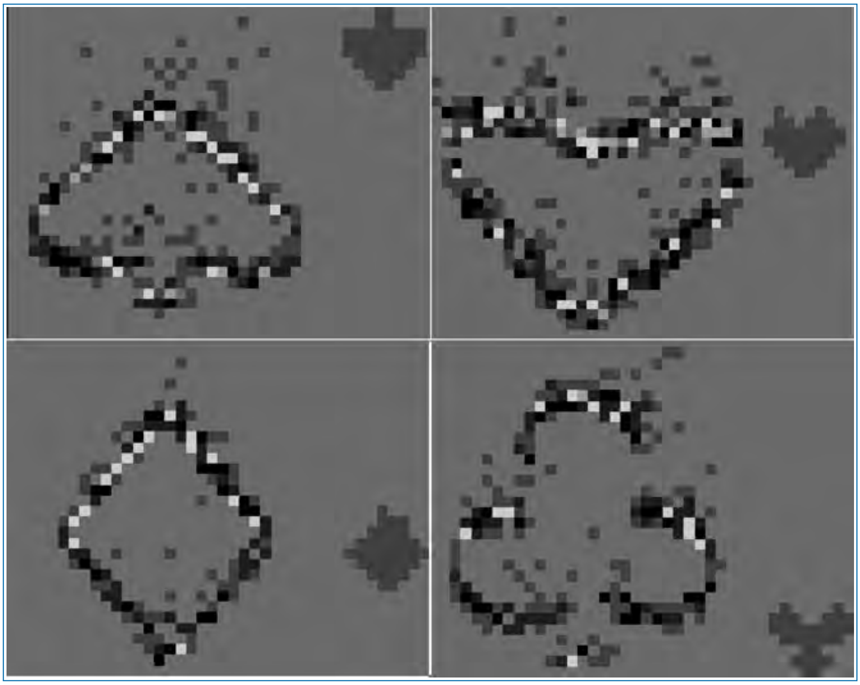


Figure 6.3. Snapshots of merging the input stimulus with the SpiNNaker classifier output. The input stimulus was generated with a 100 slow-down factor over real recording time.

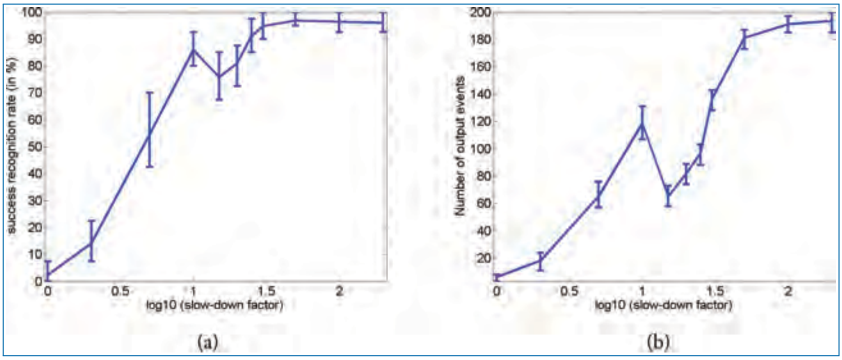


Figure 6.4. (a) Recognition rate for the sequence of 40 card symbols versus the slow-down factor of the input stimulus. (b) Total number of output events generated by the output recognition layer for the whole sequence of 40 card symbols versus the slow-down factor of the input stimulus.

timing parameters. We repeated the classification of each test sequence 30 times, measuring the classification success rate as explained above. Figure 6.4(a) shows the mean, maximum and minimum recognition success rates obtained for slow down factors [1, 2, 5, 15, 20, 25, 30, 50, 100, 200]. A 1× slow-down factor means real-time operation, that is, classification of the sequence of the high-speed browsed cards as they each pass in a 400 microsecond interval. We can observe that for slow-down factors higher than 25, the mean successful classification rate is higher than 90%. However, for slow-down factors lower than 25, the success recognition rate suffers from a severe degradation. In Figure 6.4(b), we have plotted the total number of output events generated at the output of the SpiNNaker classifier as a function of the input stimulus slow-down factor. We can observe that for slow-down factors below 50, the number of SpiNNaker output events decreases quickly. Another observation is that there is a local peak in the recognition rate (and the corresponding number of output events) at a slow-down factor of 10. For higher slow-down factors (slow-down factors of 15 and 20), the recognition rate and the number of output events in the category layer are lower.

Going into the details of the problem, we observed that the main bottleneck that limits the operation of the system is the processing time of the events in the convolution layers. We have also observed that when events are unevenly lost in subsequent layers, the spatio-temporal congruence of the recognised patterns is lost and the recognition rate decreases. This phenomenon has already been reported by Camuñas [31] who observed that queuing events in a highly saturated event processing system gives a worse performance than simply dropping them, because queuing introduces time delays, while dropping keeps the temporal coherence of the processed events. In the present case, when events are lost simultaneously in the different processing layers, the performance is better than when there is a layer that has a dominant delay. This explains the lower recognition accuracy for intermediate slow-down factors.

In Figure 6.2, we show in red numbers the total number of events that enter into the corresponding layer that have to be processed by each feature map. It can be observed that each neural population in the second convolutional layer (C3) has to process 4.7× more events/second than the first convolution layer (C1). As we have tried to maximise the number of neurons implemented per SpiNNaker core to the maximum that can be allocated, this has the downside that each core in layer C3 has to process the incoming 816,163 events in 0.95 seconds for real-time operation. As the SpiNNaker architecture is flexible, it allows us to trade-off the maximum number of neurons per core against the maximum event processing throughput.

In a first experiment, we noticed that more than one half of the weights in the first convolution layer (C1) were zero. Zero weights in the kernel add computation time per event but do not affect the result of the computation. So, we eliminated all the zero values of the kernels in the first convolution layer. In Figure 6.5, we have plotted in blue the recognition rate of the original experiment (before eliminating the zero elements in the C1 kernels) and in the green trace, we plot the recognition rate after eliminating the zero values in the C1 kernels. It can be observed that

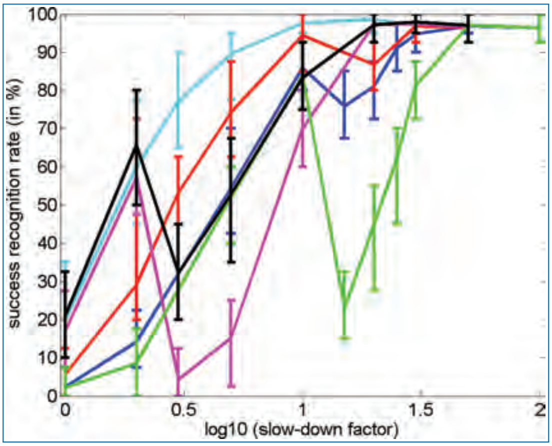


Figure 6.5. Recognition accuracy for the whole sequence of 40 card symbols versus the slow-down factor of the input stimulus when splitting each C3 neural population among several cores.

both systems perform similarly for low and high slow-down factors. However, the 'optimised' system has worse performance for intermediate slow-down factors. The reason is that by speeding-up the operation of the first C1 convolution layer, we obtain more decorrelation between the first and second convolutional layer (C3), as the second convolutional layer (C3) is the one causing the performance bottleneck in this particular case.

In a further experiment, to speed-up the operation of the second convolutional layer (C3), we mapped each neural population of layer C3 onto different SpiNNaker cores. Figure 6.5 plots the recognition rates obtained for different distributions of the feature map populations of the second convolutional layer (C3). In these experiments, we kept the elimination of the zero kernel elements in the C1 layer. In Figure 6.5, the red trace corresponds to splitting each C3 feature map operation across 2 cores. The cyan, black and magenta traces correspond to splitting each C3 feature map across 4, 5 and 6 cores, respectively. As can be observed, the 4-core division gives the optimum performance as it equalises the delays of the different layers. For further speeding up the C3 layer, the delay of the third convolutional layer (C5) becomes dominant.

6.3 Handwritten Digit Recognition with Spiking DBNs

While ConvNets continuously achieve state-of-the-arts results in computer vision tasks [93, 132, 228, 249], an alternative deep neural network architecture, known as the DBN, used to be very popular due to the fact that it can be trained on unlabelled data sets and act as a classifier as well as a generative model.

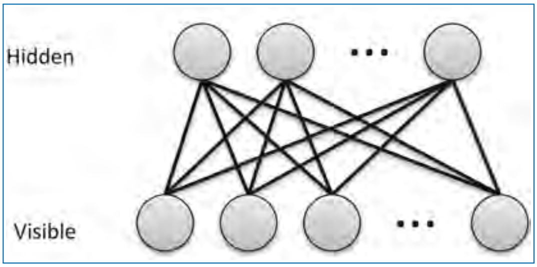


Figure 6.6. An RBM with full connectivity between visible units (bottom) and hidden units (top), but no connections within the same layer [241].

The building block of a DBN is the Restricted Boltzmann Machine (RBM). An RBM consists of two layers of neurons forming a bipartite graph (Figure 6.6). The first layer is called the 'visible' layer, and the second one is the 'hidden' layer. Each neuron in an RBM is a stochastic binary unit, whose 'active' probability is computed by the weighted sum of its inputs going through a Sigmoid activation function. DBNs are formed by stacking RBMs on top of each other, and the 'hidden' units become the 'visible' units of the next layer. Training is performed layer-by-layer, in an unsupervised manner, using the Contrastive Divergence (CD) rule [98]. The final layer is jointly trained with the input to provide a teacher signal that guides the output of the network.

O'Connor *et al.* [183] devised a method to train a DBN using the CD rule and then map the trained DBN to an SNN. They achieved this by using the Siegert approximation [122], which models a rate-based approximation of an LIF neuron firing rate, as an activation function for the RBMs. The reader should refer to the work of O'Connor *et al.* [183] for a more detailed description of the training procedure.

The SpiNNaker computing platform was used to investigate the robustness of spiking DBNs to various hardware limitations that are present in digital neuromorphic platforms such as the memory size available to store synaptic weights, the bit precision used to represent weights and neuron states, and input sensory noise commonly found in DVS sensors (silicon retinas).

The MNIST handwritten digit recognition was used as a computer vision task. The MNIST data set [138] consists of 70 k 28 × 28 greyscale images representing numbers from 0–9. The first 60 k images are used for training and the remaining 10 k for testing the trained model. Individual pixel values of the MNIST images were converted to Poisson spike trains with a rate proportional to their intensity values [183].

The research presented in this section was conducted in collaboration with the Institute of Neuroinformatics (INI) in Zurich, Switzerland and the Universite

Pierre et Marie Curie in Paris, France. A more thorough investigation of the work presented here can be found elsewhere [239–241, 243].

Spiking DBNs on SpiNNaker

The SpiNNaker computing platform was used to investigate the robustness of spiking DBNs to various hardware limitations that are present in digital neuromorphic architectures such as the limited memory available to store synaptic weights, the bit precision used to represent weights and neuron states, and the input sensor noise commonly found in DVS sensors (silicon retinas).

Porting DBNs onto SpiNNaker

For the experiments in this section, the same pre-trained DBN from O'Connor *et al.* [183] was used. This DBN consists of an input layer of 784 neurons (the 28×28 MNIST image is flattened to a vector), followed by two hidden layers of 500 neurons each and 10 output neurons, one neuron per digit. This model has in total 647,000 synapses.

After the training process is over, the DBN is mapped to an SNN by replacing the Siegert activation function with an LIF neuron model using the following equations:

$$\tau_m \frac{dV}{dt} = E_L - V + R_m I, \tag{6.1}$$

where τ_m is the membrane time constant, E_L is the resting potential and R_m is the membrane resistance. The input current I is computed as a Dirac delta synapse model,

$$I = \sum_{i=0}^{n} w_i \sum_{j=0}^{m_i} \delta(t - t_{ij}), \tag{6.2}$$

where w_i is the weight of synapse i, $\delta(t)$ is the Dirac delta function which is zero except for the firing times t_{ij} of the i^{th} neuron, n is the number of incoming synapses and m is the number of spikes that the i^{th} neuron receives. The LIF parameters used in the experiments are summarised in Table 6.2.

O'Connor *et al.* [183] used MATLAB to train and experiment with spiking DBNs. To port their trained DBN from MATLAB to SpiNNaker, a Python package was developed that generates a PyNN [44] description of the SNN ready to be executed on SpiNNaker and other SNN simulators such as Brian [81]. For the input population of a spiking DBN, the spike trains generated from each MNIST digit are described as spike arrays in PyNN using the *SpikeSourceArray* population. Additional functionality was developed for SpiNNaker that converts the spikes of a *SpikeSourceArray* to a binary file which gets uploaded to a SpiNNaker machine.

Table 6.2. Default parameters of the Leaky Integrate-and-Fire Model used in simulations [241].

Parameters	Values	Units
τ_m	5.0	s
T_{refract}	2.0	ms
V_{reset}	0.0	mV
V_{thresh}	1.0	mV

A *SpikeSourceArray* kernel on an ARM9 core in a SpiNNaker chip fetches a portion of the shared memory at every millisecond and checks which bits are set in order to generate a spike (MC packet) with the appropriate neuron ID.

Simulating Input Sensory Noise

As mentioned in the previous section, the MNIST images were converted to spike trains using a Poisson distribution with a rate proportional to their pixel intensities, while all firing rates were scaled to control the total firing rate of the input population [183]. To investigate the robustness of the spiking DBN to input sensory noise, artificial noise was introduced by redistributing a percentage of the spikes randomly across the whole input population [176]. An example of various levels of artificial noise can be seen in Figure 6.7.

Limited Weight Precision

Many digital neuromorphic platforms come with hardware constraints. Some of these limitations include a fixed number of bits for representing the weights and limited memory capacity for storing the weights [64, 162, 169, 176]. On the other hand, the weights of a trained spiking DBN are computed and stored using double floating-point precision, which according to the IEEE 754 standard has a 64-bit word length of which 52 bits are used to store the fraction, 11 bits are used for the exponent and 1 bit is used for the sign. Moreover, a Floating Point Unit (FPU) is necessary for performing floating-point computations and this translates to increased silicon area, energy costs and memory requirements for weight storage [243]. It is thus important to investigate the impact of truncating double floating-point precision weights of a trained spiking DBN to lower fixed-point precision weights.

The following notation is used: Q$m.f$, where m is the number of bits in the integer part, including the sign bit, and f is the number of bits in the fractional part. This format is a bit-level format for storing a numeric value.

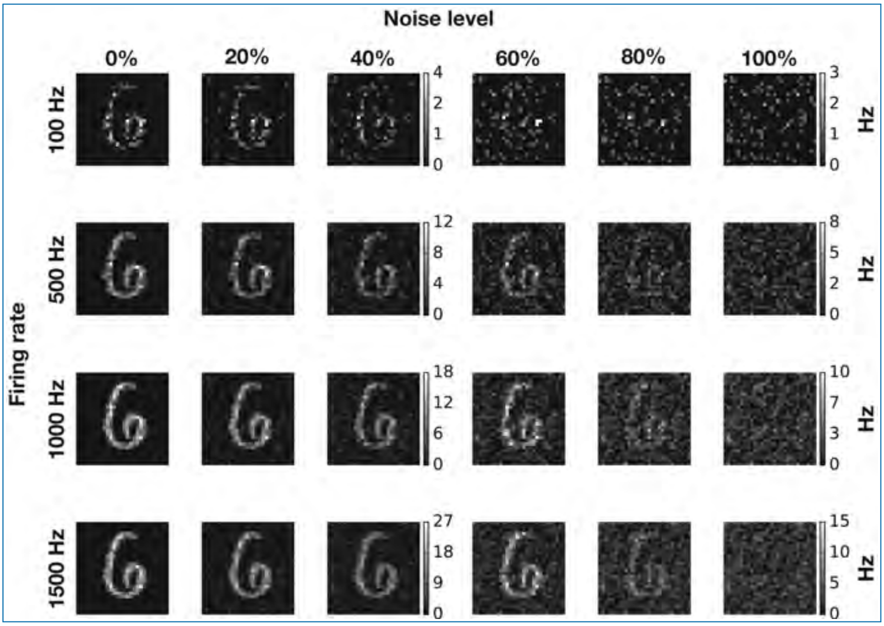


Figure 6.7. Conversion of static images to spike trains and introduction of noise. Each row represents different input rates ranging from 100 Hz to 1,500 Hz, while the columns show different percentages of input noise, from 0% up to 100%. Figure taken from [243].

The double precision floating-point weights W_H are converted to a lower-precision representation W_L using the conversion

$$W_L = \text{round}(2^f \cdot W_H) \cdot 2^{-f} \tag{6.3}$$

where W_H represent the original double floating-point weights of the trained DBN, and 2^{-f} is the resolution of the lower precision representation.

6.3.1 Results

Robustness of spiking DBNs to reduced bit precision of fixed-point synapses and input noise: This section summarises the findings of the investigation on the effect of reduced weight precision of a trained spike-based DBN and its robustness to input sensory noise.

Figure 6.8 demonstrates the effect of reduced bit precision on the trained weights of the spiking DBN of O'Connor *et al.* [183]. More specifically, Figure 6.8(a) shows the receptive fields of the first 6 neurons of the first hidden layer for different fixed-point weight resolutions. As can be visually observed, a lot of the structural information of the receptive fields is preserved, even for bit a precision of $f = 4$ bits.

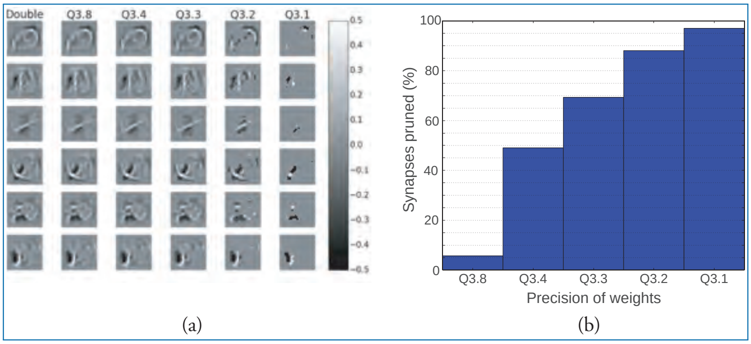
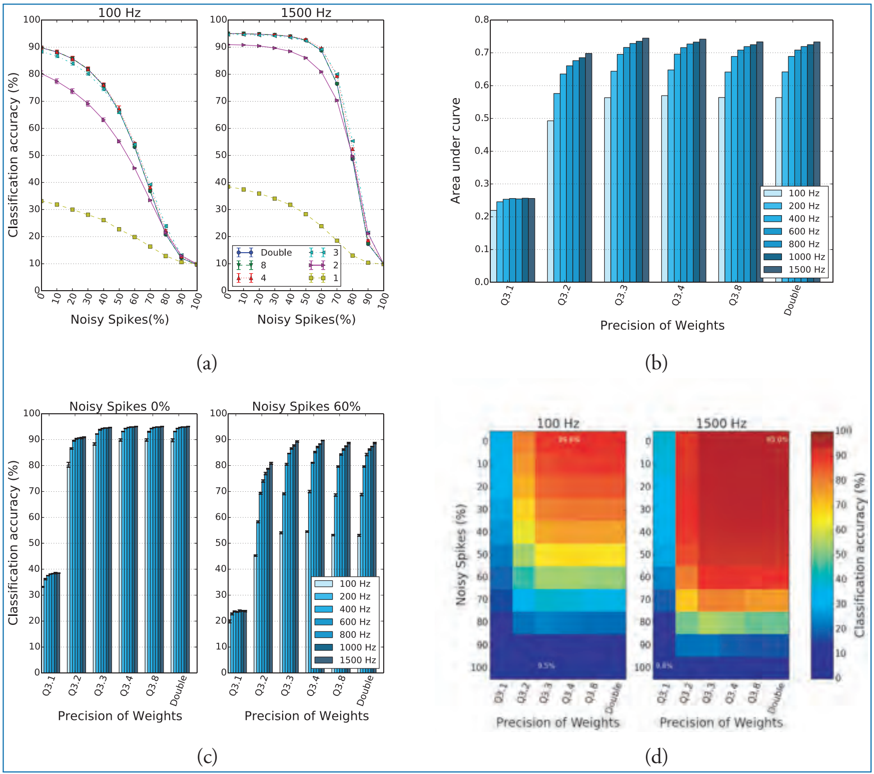


Figure 6.8. Impact of weight bit precision on the representations within a DBN. (a) The receptive fields of the first 6 neurons (rows) in the first hidden layer of the DBN with the two hidden layers. (b) Percentage of synapses from all layers that are set to zero due to the reduction in bit precision for the fractional part. Figure by Stromatias [241] with minor modifications.

Figure 6.8(b) presents the percentage of weights that were truncated to zero due to the fixed-point rounding.

Figure 6.9(a) illustrates the classification accuracy (CA) of the spiking DBN on the MNIST test set as a function of input noise and bit weight resolution for two different input firing rates (100 and 1,500 Hz), for an input stimulus of 1 second. Both curves show that the performance drops as the percentage of input noise increases, but for higher firing rates (1,500 Hz) the performances remains constant until the input noise reaches a 50% level. The peak performance stays at almost identical levels to the double floating-point precision even for bit precisions of $f = 3$. Figure 6.9(b) shows the area under the curve; a larger area translates to a higher classification performance. As in (a), a similar trend can be observed; higher input firing rates result in an increase in CA. Figure 6.9(c) demonstrates the CA for different bit weight precisions as the input firing rates increase, from 100 Hz to 1,500 Hz, for two different input noise levels, 0% and 60%. Finally, the plots in Figure 6.9(d) show that there is a wide range of input noise levels and bit weight resolutions in which the performance remains remarkably high for the two input rates, 100 Hz and 1,500 Hz. For all experiments, the performance dropped significantly when a bit weight precision of $f = 1$ was used. For a bit weight precision of $f = 2$, the CA remained approximately at 80% for 100 Hz and above 90% for firing rates higher than 600 Hz.

These findings illustrate that, indeed, the spike-based DBNs exhibit the desired robustness to input noise and numerical precision. For a weight precision of Q3.3 (6 bits per weight), the classification performance is on a par with double floating-point precision (64 bits per weight). For this particular spiking DBN,



(a)

(b)

(c)

(d)

Figure 6.9. Effect of reduced weight bit precision and input noise on the classification accuracy (CA) of the spiking DBN with two hidden layers. (a) CA as a function of input noise and bit precision of synaptic weights for two specific input spike rates of 100 and 1,500 Hz. Results over four trials. (b) Normalised area under curve in (a) for different percentages of input noise, input firing rates and weight bit precision. Higher values mean higher accuracy and better robustness to noise. (c) CA as a function of the weight bit resolution for different input firing rates and for two different noise levels, 0% and 60%. (d) CA as a 2D function of the bit resolution of the weights and the percentage of input noise for 100 Hz and 1,500 Hz input rate. The results confirm that spiking DBNs with low precision weights down to $f = 3$ bits can still reach high-performance levels and tolerate high levels of input noise. Figure by Stromatias *et al.* [243] with minor modifications.

which consists of 642,510 synapses, this means that for a weight precision of Q3.3, only 0.46 MBytes are required for storing all the weights instead of 4.9 MBytes. Moreover, one of the effects of the reduced precision is that many of the weights become zero, as seen in Figure 6.8(b), due to rounding, and thus, they can be pruned. The benefits of pruning the zeroed weights may include faster execution times due to avoiding unnecessary memory look-ups, as well as being able to execute deeper neural networks on the same hardware.

Table 6.3. Classification accuracy (CA) of the same DBN with two hidden layers running on different platforms [241].

Simulator	CA (%)	Weight precision	Description
MATLAB	96.06	Double	Rate-based (Siegert)
Brian	95.00	Double	Clock-driven
O'Connor *et al.* [183]	94.09	Double	?
SpiNNaker	94.94	Q3.8	Hybrid
Minitaur [176]	92.00	Q5.11	Event-driven

Table 6.3 summarises a comparison between the SpiNNaker platform and various hardware and software simulators, including the Brian SNN simulator, for the MNIST classification problem. The SpiNNaker results are very close to the results of the software simulation with only a 0.06% difference despite the fact that SpiNNaker uses less precise weights than standard software implementations.[2]

Classification Latency and Power Requirements of Spiking DBNs: To investigate the real-time performance of the O'Connor *et al.* [183] spiking DBN on SpiNNaker, two experiments were conducted. The first experiment investigated the mean classification latency and accuracy of the spiking DBN as a function of the number of input spikes, while the second experiment aimed at measuring the mean classification latency of the spike-based DBN running on SpiNNaker. An additional experiment was performed to investigate the power requirements of the spiking DBN running on a single SpiNNaker chip as a function of the input firing rate.

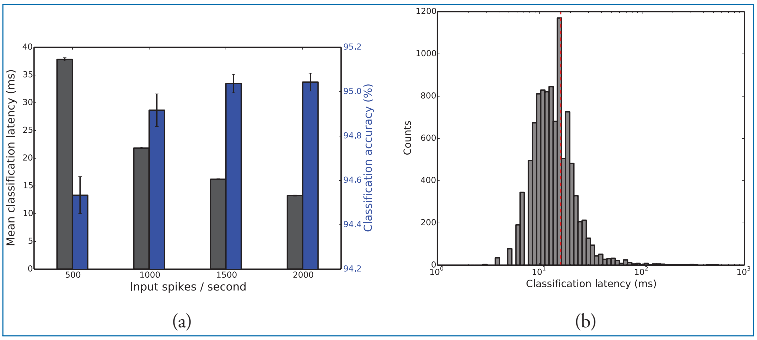
For the experiments described in this section, the following SpiNNaker configuration was used: the ARM9 processor clocks were set to 200 MHz, the routers and system busses were set to 100 MHz, while the off-chip memory clocks were set to 133 MHz.

For the first experiment, the static images of the MNIST test digits were converted to spike trains with input firing rates ranging from 500 Hz up to the point where additional input spikes per second had no effect on the mean classification accuracy. Each experiment was executed for four trials and results were averaged across all trials.

Results showed that for the spiking DBN, increasing the input firing rate reduces the mean classification latency as observed in Figure 6.10(a). More specifically, when the input firing rate is 1,500 Hz, the mean classification latency becomes

[2] A video of a spiking DBN running on SpiNNaker and recognising a handwritten digit can be seen here: https://youtu.be/f-Xi2Y4TB58

16.2 ms (Figure 6.10(b)), while the classification accuracy is 95.0%. For firing rates above 1,500 Hz, there is no effect on the mean classification accuracy; however, increasing the input firing rate to 2,000 Hz reduces the mean classification latency to 13.2 ms. What can also be observed from Figure 6.10(a) is that increasing the total number of input spikes reduces the standard deviation for both the mean classification latency and the classification accuracy.



(a) (b)

Figure 6.10. (a) Mean classification latency and classification accuracy as a function of the input spikes per second for the spiking DBN. (b) Histogram of the classification latencies for the MNIST digits of the testing set when the input rates are set to 1,500 Hz. The mean classification latency of the DBN with two hidden layers is 16 ms [240]. Figure by Stromatias [241] with minor modifications.

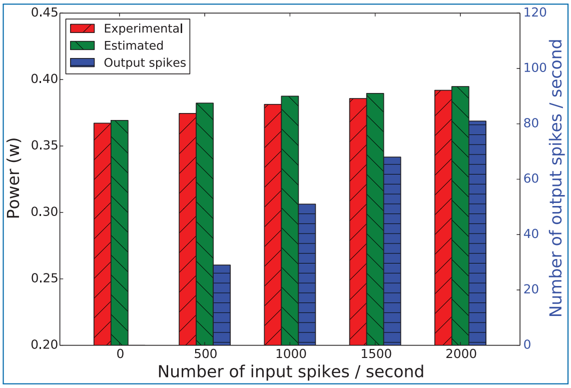


Figure 6.11. Real and estimated power dissipation of the O'Connor *et al.* [183] spike-based DBN running on a single SpiNNaker chip as a function of the number of input spikes generated for the same MNIST digit. The right axis shows the number of output spikes as a function of the number of input spikes. The left bars (0 input spikes) show power dissipation when the network is idle. The model used to estimate the power dissipation of SNN running on a SpiNNaker machine is based on the work of [240, 242]. Figure by Stromatias [241].

Finally, Figure 6.11 shows the power requirements of the O'Connor *et al.* [183] spiking DBN running on a single SpiNNaker chip. Results show that when an input firing rate of 2,000 Hz is used per digit, a single SpiNNaker chip dissipates 0.39 W. That accounts for simulating 1,794 LIF neurons with an activity of 1,569,000 synaptic events (SE) per second. For the identical spiking DBN implemented on Minitaur, an FPGA event-driven SNN simulator clocked at 75 MHz, a power dissipation of 1.5 W was reported for 1,000 spikes per image [176].

6.4 Spiking Deep Neural Networks

An intuitive idea for bringing these deep learning techniques to SNNs is either to transform well-tuned deep ANN models into SNNs or to translate numerical calculations of weight modulations into biologically plausible synaptic learning rules. Based on the former approach, this section proposes, based on the work of Liu [146], a generalised method to train SNNs off-line on equivalent ANNs and transfer the tuned weights back to the SNNs. There are two significant problems to be solved when training SNNs off-line. First, an accurate activation function is needed to model the neural dynamics of spiking neurons. In this section, we propose a novel activation function used in ANNs, Noisy Softplus (NSP), to closely simulate the firing activity of LIF neurons driven by noisy current influx. The second problem is to map the abstract numerical values of the ANNs to physical variables, current (nA) and firing rate (Hz), in the SNNs. Consequently, we introduce the Parametric Activation Function (PAF) $y = p \times f(x)$, which successfully associates physical units with conventional activation functions and thus unifies the representations of neurons in ANNs and the ones in SNNs. Therefore, an SNN can be modelled and trained on an equivalent ANN using conventional training algorithms, such as backpropagation.

The significance lies in the simplicity and generalisation of the proposed method. SNN training, now, can be simplified to: firstly, estimate parameters for the PAF using NSP; secondly, use the PAF version of conventional activation functions to train an equivalent ANN; and finally, transfer the tuned weights directly into the SNN without any conversion. Regarding the generalisation, it works exactly the same as training ANNs: the same feed-forward network architecture, backpropagation algorithm and activation functions, and uses the most common spiking neurons, standard LIF, that run on most neuromorphic hardware platforms.

Therefore, most importantly, this research provides the neuromorphic engineering community with a simple, but effective and generalised off-line SNN training method which notably simplifies the development of AI applications on neuromorphic hardware. In turn, it enables ANN users to implement their models on neuromorphic hardware without the knowledge of spiking neurons or programming specific hardware, thereby enabling them to benefit from the advantages of neuromorphic computers: such as real-time processing, low latency, biological realism and energy efficiency. Furthermore, the success of the proposed off-line training method paves the way to energy-efficient AI on neuromorphic machines scaling from mobile devices to huge computer clusters.

6.4.1 Related Work

A regular artificial neuron comprises a weighted summation of input data, $net_j = \sum x_i w_i$, and an activation function, f, applied to the sum, net_j. However the inputs of a spiking neuron (Figure 6.12) are spike trains, which generate current influx through neural synapses (connections). A single spike creates a current pulse with an amplitude of w, which is defined as the synaptic efficacy, and the current then decays exponentially with a decay rate determined by the synaptic time constant, τ_{syn}. The current pulses consequently produce PSPs on the neuron driving its membrane potential to change over time and trigger spikes as outcomes when the neuron's membrane potential reaches some threshold. The dynamics of the current influxes, PSPs, membrane potentials and spike trains are all time dependent, while the neurons of ANNs only cope with abstract numerical values representing spiking rate, without timing information. Therefore, these fundamental differences in input/output representation and neural computation form the main research problem of how to operate and train biologically plausible SNNs to make them as competent as ANNs in cognitive tasks. In this section, we focus on the solutions of off-line training where SNNs are trained on equivalent ANNs and then the tuned weights are transferred to the SNNs.

Jug *et al.* [122] first proposed the Siegert formula [226] to model the response function of a spiking neuron, which worked as a Sigmoid unit in training spiking

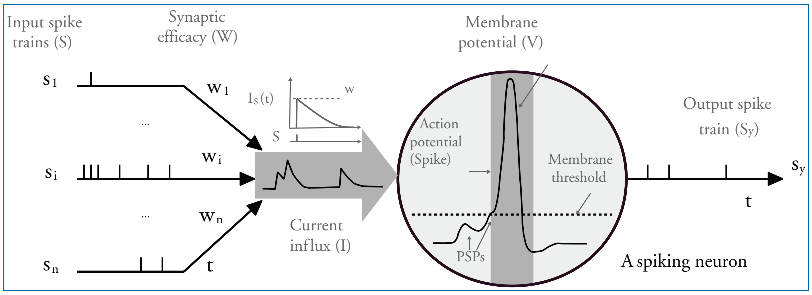


Figure 6.12. A spiking neuron. Spike trains flow into a spiking neuron as current influx, trigger linearly summed PSPs through synapses with different synaptic efficacy **w**, and the post-synaptic neuron generates output spikes when the membrane potential reaches some threshold.

deep belief networks. The Siegert formula maps incoming currents driven by Poisson spike trains to the response firing rate of a LIF neuron, similar to the activation functions used in ANNs which transform the summed input to corresponding outcomes. The variables of the response function are in physical units, and thus, the trained weights can be transferred directly into SNNs.

However, the Siegert formula is inaccurate as it models the current noise as white [147], $\tau_{syn} \to 0$, which is not feasible in practice.

Moreover, the high complexity of the Siegert function and the computation of its derivative to obtain the error gradient cause much longer training times, thus consuming more energy, when compared to regular ANN activation functions, for example, Sigmoid. We will illustrate these problems in detail in Section 6.4.2.

A softened version of the response function of LIF neurons has been proposed [111] and is less computationally expensive than the Siegert function. However, the model ignores the dynamic noise change introduced by input spikes, assuming a static noise level of the current influx. Therefore, the training requires additional noise on the response firing rate and on the training data; however, the manually added noise is far from the actual activity of the network and includes hyper-parameters in the model.

Although the trained weights can be directly used in SNNs since both the above LIF response functions accept and output variables in physical units, they struggle in terms of poor modelling accuracy and high computational complexity. Moreover, they lose the numerical abstraction of firing rates in ANNs, thus, being constrained to SNN training. Meanwhile, other widely used activation functions in ANNs cannot be transformed to model SNNs. Therefore, the first problem is the accurate modelling of the neural response activity of LIF neurons using abstract activation functions, in the hope of (1) increasing the modelling accuracy, (2) reducing the computation complexity and (3) generalising off-line SNN training to commonly used ANN activation functions. These activation functions used in ANNs without physical units are called 'abstract' to differ from the response functions of spiking neurons. We select them for LIF modelling because of the simplicity and generalised use for training ANNs. Thus, we propose the activation function, NSP [147], in Section 6.4.2 to address this problem.

Then, the second problem is to map the abstract activation functions to physical units used in SNNs: current in nA and firing rates in Hz. In doing so, the neuronal activities of an SNN can be modelled with such scaled activation functions and the trained weights can be transferred into SNNs without conversion. Instead of directly solving this problem, an alternative way is to train an ANN with abstract activation functions and then modulate the trained weights to fit in SNNs. Researchers [32, 51] successfully applied this method on less biologically realistic and simplified integrate-and-fire (IF) neurons. Nevertheless, these simple

IF neurons are usually difficult to implement in analogue circuits, and thus they are feasible only on digital neuromorphic hardware, for example, TrueNorth [163]. Tuning these trained ANN models to adapt to simplified IF neurons is relatively straightforward, so this method sets the state-of-the-art performance. However, this section (in Section 6.4.3) aims to address the second problem of mapping abstract activation functions to the response firing activity of biologically plausible LIF neurons. Thus, not only the training can be simplified by using conventional simple activation functions, such as Rectified Linear Units (ReLUs), but also the method can be generalised to target standard LIF neurons which are supported by most neuromorphic hardware.

6.4.2 Siegert: Modelling the Response Function

The response function, in the context of this section, indicates the firing rate of a spiking neuron in the presence of input current. In this section, we introduce the first use of the Siegert formula to model the response function of a LIF neuron. Although the Siegert formula enables off-line SNN training, it has several drawbacks. Therefore, we propose the first abstract activation function, NSP, to model the LIF response function.

Biological Background

A LIF neuron model is as follows:

$$\tau_m \frac{dV}{dt} = V_{rest} - V + R_m I(t). \tag{6.4}$$

The membrane potential V changes in response to the input current I, starting at the resting membrane potential V_{rest}, where the membrane time constant is $\tau_m = R_m C_m$, R_m is the membrane resistance and C_m is the membrane capacitance. The central idea in converting spiking neurons to activation units lies in the response function of a neuron model. Given a constant current injection I, the response function, that is, firing rate of the LIF neuron is:

$$\lambda_{out} = \left[\tau_{refrac} - \tau_m \log\left(1 - \frac{V_{th} - V_{rest}}{IR_m}\right)\right]^{-1}, \quad \text{when } IR_m > V_{th} - V_{rest}, \tag{6.5}$$

otherwise, the membrane potential cannot reach the threshold V_{th} and the output firing rate is zero. The absolute refractory period τ_{refrac} is included, during which period synaptic inputs are ignored.

However, in practice, a noisy current generated by the random arrival of spikes, rather than a constant current, flows into the neurons. The noisy current is typically treated as a sum of a deterministic constant term, I_{const}, and a white noise term, I_{noise}. Thus, the value of the current is Gaussian distributed with m_I mean and ${s_I}^2$ variance. The white noise is a stochastic process $\xi(t)$ with mean 0 and variance 1, which is delta-correlated, that is, the process is uncorrelated in time so that a value $\xi(t)$ at time t is totally independent on the value at any other time t'. Therefore, the noisy current can be seen as:

$$I(t) = I_{const}(t) + I_{noise}(t) = m_I + s_I\xi(t), \tag{6.6}$$

and accordingly, Equation 6.4 becomes:

$$\frac{dV}{dt} = \frac{V_{rest} - V}{\tau_m} + \frac{m_I}{C_m} + \frac{s_I}{C_m}. \tag{6.7}$$

We then multiply both sides of Equation 6.7 by a short time step dt. The stochastic differential equation of the membrane potential becomes:

$$\begin{aligned} dV &= \frac{V_{rest} - V}{\tau_m}dt + \frac{m_I}{C_m}dt + \frac{s_I}{C_m}dW_t \\ &= \frac{V_{rest} - V}{\tau_m}dt + \frac{m_I}{C_m}dt + \frac{s_I\sqrt{dt}}{C_m}r \\ &= \frac{V_{rest} - V}{\tau_m}dt + \mu_v dt + \sigma_v r. \end{aligned} \tag{6.8}$$

The last term dW_t is a Wiener process; $W_{t+dt} - W_t$ obeys a Gaussian distribution with mean 0 and variance dt; thus, r is a random number sampled in accordance with a Gaussian distribution of zero mean and unit variance. The instantaneous mean μ_v and variance ${\sigma_v}^2$ of the change in membrane potential characterise the statistics of V in a short time range, and they can be derived from the statistics of the noisy current:

$$\mu_v = \frac{m_I}{C_m}, \qquad \sigma_v = \frac{s_I\sqrt{dt}}{C_m}. \tag{6.9}$$

The response function [134, 206] of the LIF neuron to a noisy current, also known as the Siegert formula, is a function of μ_v and σ_v:

$$\lambda_{out} = \left[\tau_{refrac} + \tau_m \int_{\frac{V_{rest} - \mu_v \tau_m}{\sigma_v \sqrt{\tau_m}}}^{\frac{V_{th} - \mu_v \tau_m}{\sigma_v \sqrt{\tau_m}}} \sqrt{\pi} \exp(V^2)(1 + erf(V))dV \right]^{-1}. \tag{6.10}$$

Figure 6.13 shows the response curves (Equation 6.10) of a LIF neuron driven by noisy currents where the Gaussian noise is of mean m_I and standard deviation s_I. The parameters of the LIF neuron are all biologically plausible (see the listed values in Table 6.4), and the same parameters are used throughout this chapter.

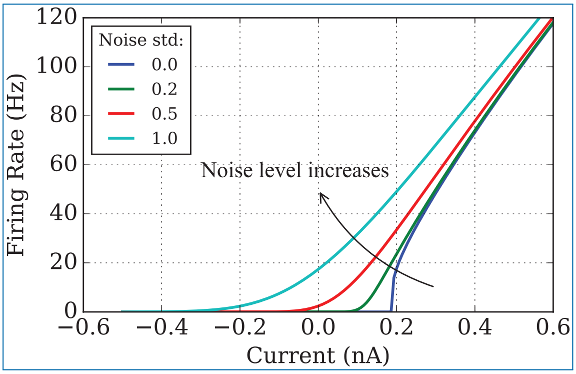


Figure 6.13. Response function of the LIF neuron with noisy input currents with different standard deviations.

Table 6.4. Parameter setting for the current-based LIF neurons using PyNN.

Parameters	Values	Description
C_m	0.25 nF	Membrane capacitance
τ_m	20.0 ms	Membrane time constant
τ_{refrac}	1.0 ms	Refractory period
V_{reset}	−65.0 mV	Resting membrane potential
V_{rest}	−65.0 mV	Resetting membrane potential
V_{th}	−50.0 mV	Membrane threshold
I_{offset}	0.0 nA	Offset of current influx

The bottom (zero noise) line in Figure 6.13 illustrates the response function of such a LIF neuron injected with constant current, which inspired the proposal of ReLUs. As noise increases, the level of firing rates also rises. Thus, the Softplus function approximates the response activity to noisy current, but only represents a single level of noise; for example, the top line in Figure 6.13 shows the curve when $s_I = 1$.

Mismatch of the Siegert Function to Practice

To verify the Siegert function in practice, simulation tests were carried out using PyNN [44] to compare the reality with the analytical results (the Siegert function). The noisy current was produced by *NoisyCurrentSource* in PyNN which is a constant current of m_I added to a Gaussian white noise of zero mean and s_I^2

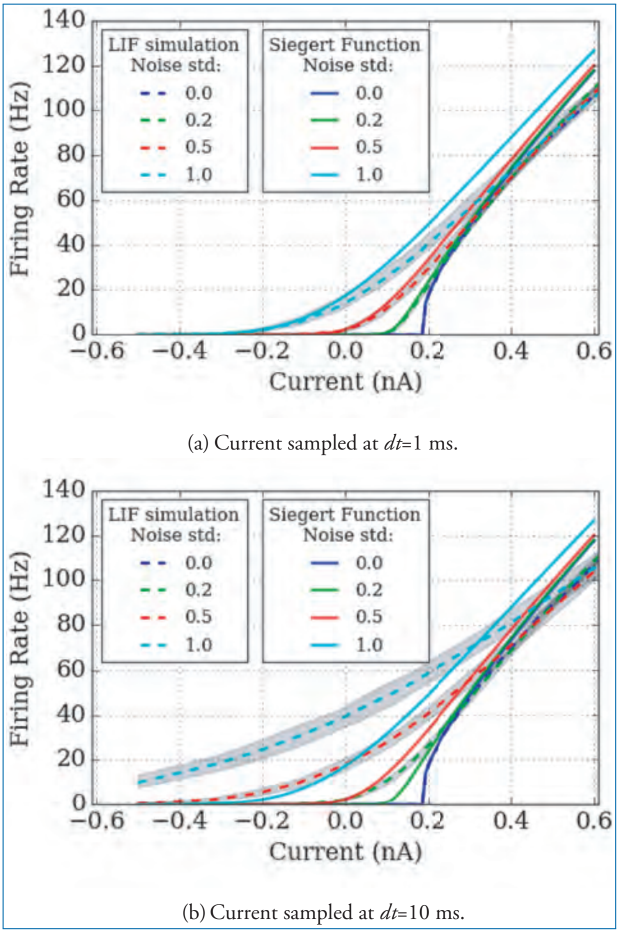
variance. The noise was drawn from the Gaussian distribution in a time resolution of dt. We chose $dt = 1$ ms and $dt = 10$ ms for comparison. For a given pair of m_I and s_I^2, a noisy current was injected into a current-based LIF neuron for 10 s, and the output firing rate was the average over 10 trials. There were four noise levels tested in the experiments: 0, 0.2, 0.5, 1; and the mean current increased from −0.5 to 0.6 nA.

The dashed curves in Figures 6.14 illustrate the output firing rate of the LIF simulations, while the bold lines are the analytical reference, the Siegert function (the same as in Figure 6.13). The differences between the practical simulations and the Siegert function enlarge when the time resolution, dt, of the *NoisyCurrentSource* increases from 1 ms (Figure 6.14(a)) to 10 ms (Figure 6.14(b)). The sampled current signals (*NoisyCurrentSource*) are shown in Figure 6.15(a) and (b). The discrete sampling of the noisy current introduces time step correlation to the white noise, shown in Figure 6.15(e) and (f), where the value remains the same within a time step dt. Although both current signals follow the same Gaussian distribution, see Figure 6.15(g) and (h), the current is approximately a white noise when $dt = 1$ ms, but a coloured noise, for example, increases Power Spectral Density (PSD) at lower frequency, when $dt = 10$ ms, see Figure 6.15(c) and (d). Therefore, the coloured noise of the current influx drives the LIF neuron to fire observably more intensely. Hence, the Siegert formula, Equation 6.10, can only approximate the LIF response of noisy current with white noise, but it is not adapted to coloured noise. In practice, the current is generated by random arrivals of input spikes with various synaptic efficiencies, which also brings in coloured noise.

A more realistic simulation of a noisy current can be generated by 100 Poisson spike trains, where the mean and variance of the current are given by La Camera *et al.* [134]:

$$m_I = \tau_{syn} \sum_i w_i \lambda_i, \quad s_I^2 = \frac{1}{2} \tau_{syn} \sum_i w_i^2 \lambda_i, \tag{6.11}$$

where τ_{syn} is the synaptic time constant, and each Poisson spike train connects to the neuron with a strength of w_i and a firing rate of λ_i. Two populations of Poisson spike sources, for excitatory and inhibitory synapses respectively, were connected to a single LIF neuron to mimic the noisy currents. The firing rates of the Poisson spike generators were determined by the given m_I and s_I. Figure 6.16 illustrates the recorded firing rates responding to the Poisson spike trains compared to the mean firing rate driven by *NoisyCurrentSource* in Figure 6.14. Note that the estimation of LIF response activity using the Siegert function requires noisy current with white noise; however, in practice the release of neurotransmitter takes time ($\tau_{syn} \gg 0$) and the synaptic current decays exponentially $I_{syn} = I_0 e^{-\frac{t}{\tau_{syn}}}$. Figure 6.17(a) and (b)

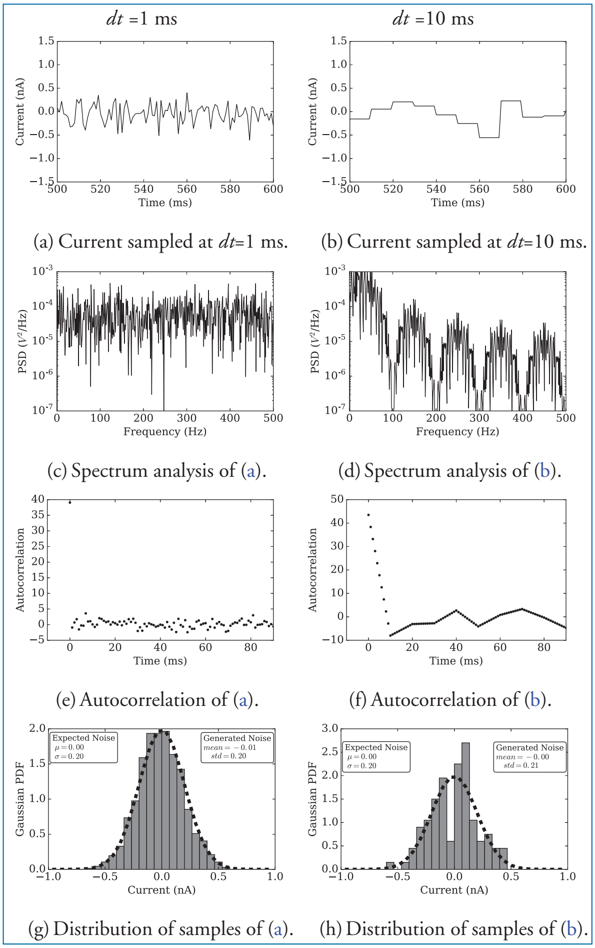


(a) Current sampled at *dt*=1 ms.

(b) Current sampled at *dt*=10 ms.

Figure 6.14. Recorded response firing rate of a LIF neuron driven by *NoisyCurrentSource* in PyNN, compared to the Siegert function. The *NoisyCurrentSource* is sampled at every (a) 1 ms and (b) 10 ms. Averaged firing rates of 10 simulation trails tested on four noisy levels are shown in different colours of dashed lines, and the grey colour fills the range between the minimum to maximum of the firing rates. The analytical LIF response function, the Siegert formula (Equation 6.10), is drawn in bold lines (shown in Figure 6.13) to compare with the practical simulations. The selected noise levels are the same with the LIF simulations, and the curves are plotted with the same colours as the dashed lines.

shows two examples of synaptic current of 0 nA mean and 0.2 standard deviation driven by 100 neurons firing at the same rate and with the same synaptic strength (half excitatory, half inhibitory), but of different synaptic time constant. Therefore, the current at any time *t* during the decay period is dependent on the value



(a) Current sampled at dt=1 ms.

(b) Current sampled at dt=10 ms.

(c) Spectrum analysis of (a).

(d) Spectrum analysis of (b).

(e) Autocorrelation of (a).

(f) Autocorrelation of (b).

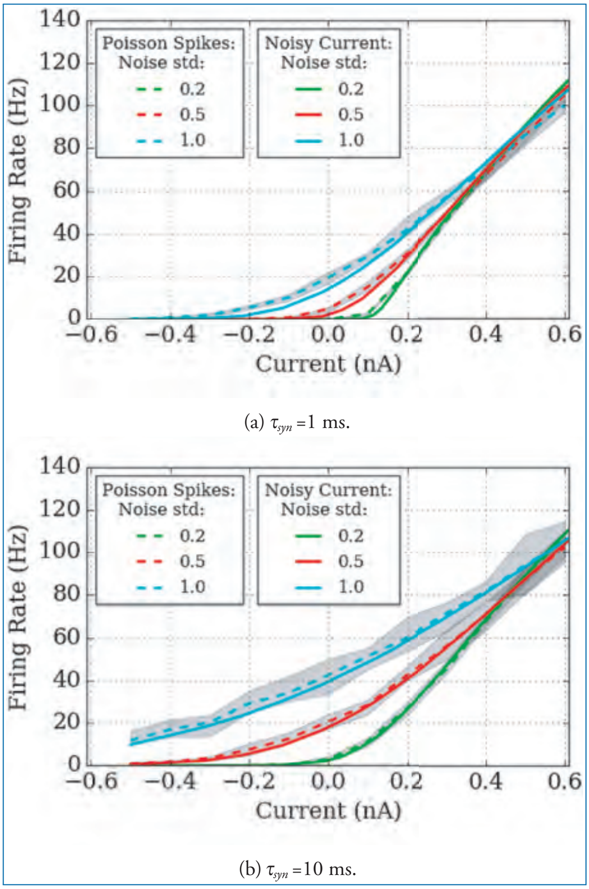
(g) Distribution of samples of (a).

(h) Distribution of samples of (b).

Figure 6.15. *NoisyCurrentSource* samples noisy currents from a Gaussian distribution every 1 ms (left) and 10 ms (right). The signals are shown in the time domain in (a) and (b), and in the spectrum domain in (c) and (d). The autocorrelation of both current signals are shown in (e) and (f). The distribution of the discrete samples are plotted in bar charts to compare with the PDF of the original Gaussian distribution, shown in (g) and (h).

at the previous time step, which makes the synaptic current a coloured noise, see Figure 6.17(c) and (d).

We observe in Figure 6.16(a) that the response firing rate to synaptic current is higher than the *NoisyCurrentSource* for most of the current range. This is caused by the coarse resolution (1 ms) of the spikes, and thus, the standard deviation of the current is larger than 0.2, shown in Figure 6.17(g); and the τ_{syn}, even when as short as 1 ms, adds coloured noise instead of white noise to the current. However, Figure 6.16(b) shows a similar firing rate of both the synaptic driven current and

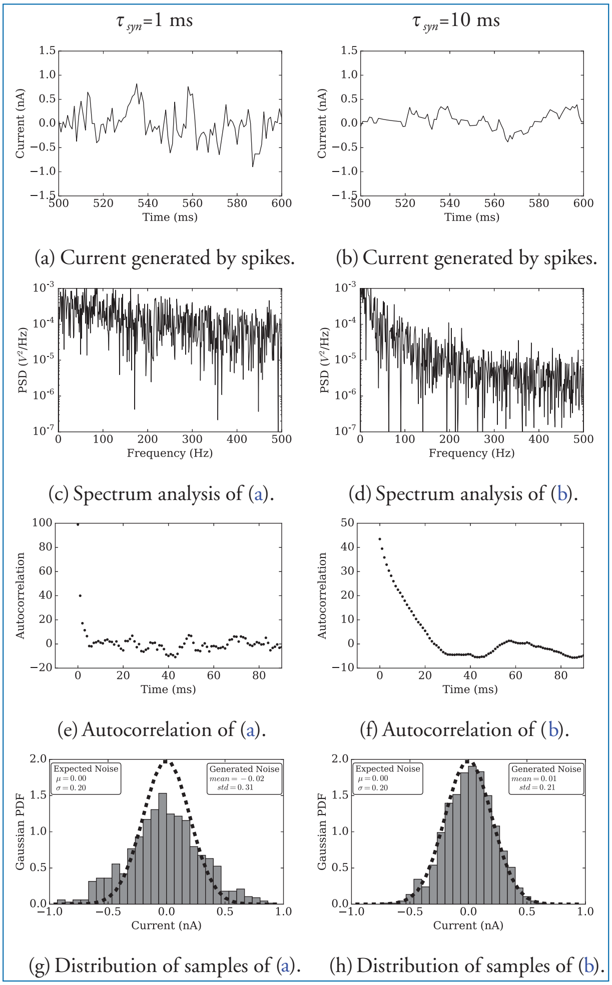


(a) τ_{syn} =1 ms.

(b) τ_{syn} =10 ms.

Figure 6.16. Recorded response firing rate of a LIF neuron driven by a noisy synaptic current, which is generated by random arrivals of Poisson spike trains, compared to previous experiments using *NoisyCurrentSource*. Averaged firing rates of 10 simulation trails tested on three noisy levels are shown in different colours of dashed lines, and the grey colour fills the range between the minimum to maximum of the firing rates. The other LIF simulation using *NoisyCurrentSource* is drawn in bold lines (same as the dashed lines in Figure 6.14) to compare with the noisy synaptic current. The same noise level is plotted with the same colour for both experiments. Two synaptic time constants are tested: (a) $\tau_{syn} = 1$ ms, to compare with *NoisyCurrentSource* sampled at every 1 ms, and (b) $\tau_{syn} = 10$ ms, to compare with *NoisyCurrentSource* sampled at every 10 ms.

the *NoisyCurrentSource*, since both of the current signals have similar distribution (Figure 6.17(h)) and time correlation (Figure 6.17(f)). Nevertheless, the analytical response function, the Siegert formula, cannot approximate either of the practical simulations (see Figure 6.14).



(a) Current generated by spikes. (b) Current generated by spikes.

(c) Spectrum analysis of (a). (d) Spectrum analysis of (b).

(e) Autocorrelation of (a). (f) Autocorrelation of (b).

(g) Distribution of samples of (a). (h) Distribution of samples of (b).

Figure 6.17. Noisy currents generated by 100 Poisson spike trains to a LIF neuron with synaptic time constant $\tau_{syn} = 1$ ms (left) and $\tau_{syn} = 10$ ms (right). The currents are shown in the time domain in (a) and (b), and in the spectrum domain in (c) and (d). The autocorrelation of both current signals are shown in (e) and (f). The distribution of the generated samples is plotted in bar chart form to compare to the expected Gaussian distribution, shown in (g) and (h).

Although the use of the Siegert function opened the door for modelling the LIF response function to work similarly to the activation functions used in ANNs [122], there are several drawbacks of this method:

- Most importantly, the numerical analysis of an LIF response function is far from accurate in practice. 'Practice' here means SNN simulations using LIF neurons. Thus, the inaccurate model generates errors between the estimation and the practical response firing rate.

- The high complexity of the Siegert function causes much longer training times and more energy, let alone the high-cost computation on its derivative.
- The Siegert function is used to replace Sigmoid functions for training spiking RBMs [122]. Therefore, neurons have to fire at high frequency (higher than half of the maximum firing rate) to represent the activation of a sigmoid unit; thus, the network activity results in high power dissipation.
- Better learning performance has been reported using ReLU rather than Sigmoid units, so modelling spiking neurons with a ReLU-like activation function is needed.

Therefore, we propose the NSP function which provides solutions to the drawbacks of the Siegert unit.

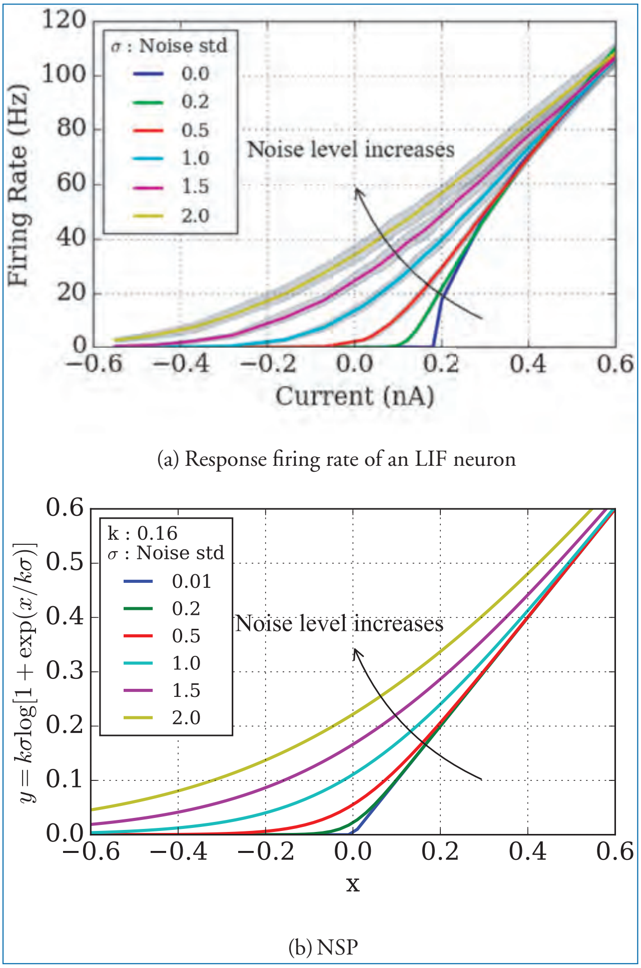
Noisy Softplus (NSP)

Due to the limited time resolution of common SNN simulators and the time taken for neurotransmitter release, τ_{syn}, mismatches exist between the analytical response function, the Siegert formula and practical neural activities. Consequently to model the practical LIF response function (see Figure 6.18(a)) whose output firing rates are determined by both the mean and variance of the noisy input currents, the NSP is proposed as follows:

$$y = f_{NSP}(x, \sigma) = k\sigma \log\left[1 + \exp\left(\frac{x}{k\sigma}\right)\right], \tag{6.12}$$

where x and σ refer to the mean and standard deviation of the input current, y indicates the intensity of the output firing rate and k, determined by the biological configurations on the LIF neurons [147] (listed in Table 6.4), scales the impact of the noise thereby controlling the shape of the curves. Note that the novel activation function we proposed contains two parameters, the mean current and its noise, which takes the values estimated by Equation 6.11: m_I and s_I^2. Since the NSP takes two variables as inputs, the activation function can be plotted in 3D, see Figure 6.19.

Figure 6.18(b) shows the activation function in curve sets corresponding to different discrete noise levels which mimic the responses of practical simulations of LIF neurons, shown in Figure 6.18(a). It is noteworthy that the non-smooth curve (blue line in Figure 6.18(a) generated by $\sigma = 0$) of the LIF response activities does not fit the NSP function; this is a limitation of using NSPs to model spike rates when the noise level approaches 0. However, we ignore this minor mismatch to unify and simplify the model, since the results show an acceptable performance drop in Section 6.4.4. In addition, scaling, shifting and parameter calibrations are



(a) Response firing rate of an LIF neuron

(b) NSP

Figure 6.18. NSP models the LIF response function. (a) Firing rates measured by simulations of a LIF neuron driven by different input currents and discrete noise levels. Bold lines show the average and the grey colour fills the range between the minimum and the maximum. (b) NSP activates the input x according to different noise levels where the noise scaling factor $k = 0.16$.

essential to fit the NSP accurately to LIF responses. We will illustrate the procedure in Section 6.4.3.

The derivative of the NSP is the logistic function scaled by $k\sigma$:

$$\frac{\partial f_{NSP}(x,\sigma)}{\partial x} = \frac{1}{1 + \exp\left(-\frac{x}{k\sigma}\right)}, \tag{6.13}$$

which could be applied easily to back propagation in ANN training.

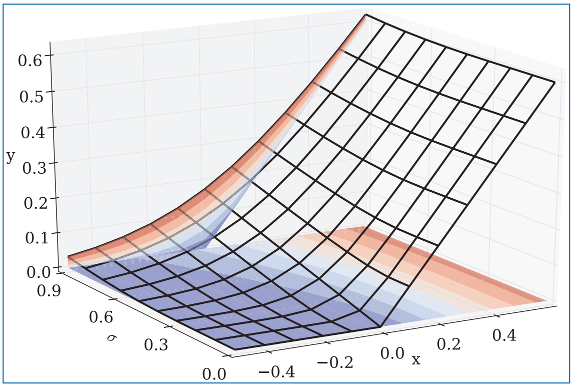


Figure 6.19. Noisy Softplus in 3D.

6.4.3 Generalised Off-line SNN Training

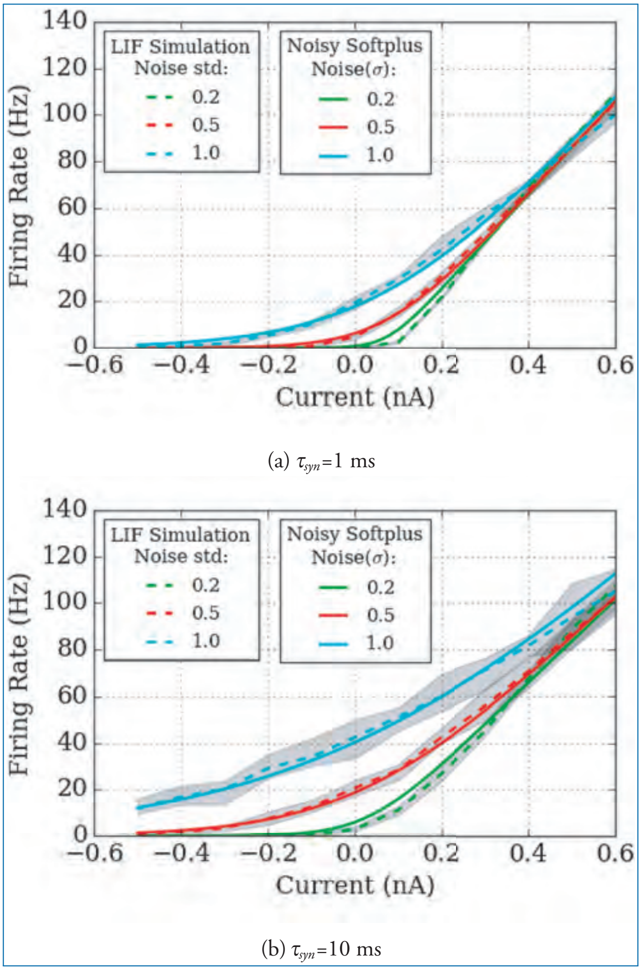
We have briefly discussed modelling the response of an LIF neuron with an abstract activation function, NSP. Function transformations are essential to map the numerical values of NSP precisely to physical variables in SNNs. Therefore, to optimise the modelling of the LIF response function, we curve fit parameters of the NSP (e.g. Figure 6.18(b)) to approximate the actual firing activities (e.g. Figure 6.18(a)). This section includes the parameters in the proposed activation function, PAF, to unify the presentation of activation functions for both ANNs and SNNs. Thus, a generalised off-line SNN training method is completed using PAF. Moreover, a corresponding fine tuning method is put forward to increase the training capability.

Mapping NSP to Concrete Physical Units

The inputs of the NSP function, x and σ, are obtained from physical variables as stated in Equation 6.11: m_I and s_I, and thus, they are already in physical units (nA). Therefore, linearly scaling up the activation function by a factor S (Hz / nA) can approximate the output firing rate λ_{out} of an LIF neuron in Hz. Moreover, including a bias b on the input x allows the curve set to move freely on the x-axis to fit the actual firing activities better:

$$\lambda_{out} \simeq f_{NSP}(x - b, \sigma) \times S = k\sigma \log\left[1 + \exp\left(\frac{x - b}{k\sigma}\right)\right] \times S. \tag{6.14}$$

Suitable calibrations of the noise scaling factor k, input bias b and mapping scaling factor S in Equation 6.14 enable NSP to match closely the practical response firing rates of LIF neurons given various biological parameters. The parameters (k, b, S) are curve-fitted with the triple data points of $(\lambda_{out}, x, \sigma)$ and the calibration currently is conducted by linear least squares regression. The output firing rate



(a) τ_{syn}=1 ms

(b) τ_{syn}=10 ms

Figure 6.20. NSP fits to the response firing rates of LIF neurons in concrete physical units. Averaged firing rates of 10 simulation trails tested on three noisy levels are shown in different colours of dashed lines, and the grey colour fills the range between the minimum to maximum of the firing rates (same as the dashed lines in Figure 6.16). The bold lines are the scaled NSP, where the same noise level is plotted with the same colour as the LIF simulations. The parameters used in the experiments are as follows: (a) $\tau_{syn} = 1$ ms for LIF simulation, and $k = 0.18$, $S = 201.66$, $b = 0.07$ for NSP; (b) $\tau_{syn} = 10$ ms for LIF simulation, and $k = 0.35$, $S = 178.91$, $b = 0.03$ for NSP.

λ_{out} is measured from SNN simulations where an LIF neuron is driven by synaptic input currents of Poisson spike trains, and x and σ take the mean and variance of the noisy current using Equation 6.11. Figure 6.20 shows two calibration results in which the parameters were fitted to $(k, b, S) = (0.18, 0.07, 201.66)$

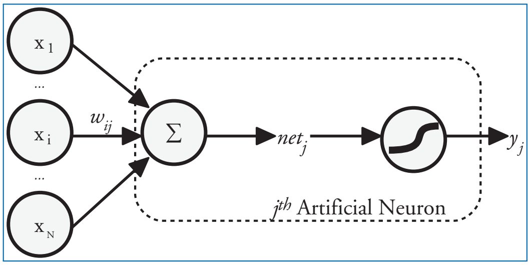


Figure 6.21. A general artificial neuron where an activation function transforms the weighted sum net_j to its outcome y_j.

when the synaptic constant is set to $\tau_{syn} = 1$ ms and was fitted to $(k, b, S) = (0.35, 0.03, 178.91)$ when $\tau_{syn} = 10$ ms.

To keep the simple format of traditional activation functions, $y = f(x)$, which has no constant bias on the input, it is easy to pass the bias b to the LIF parameter, the constant current offset, $I_{offset} = b$. Therefore, the specific parameter I_{offset} of the LIF neuron is not chosen arbitrarily, but configured by precise estimation of b. More importantly, setting I_{offset} properly on the LIF neuron instead of having a constant bias on the input of an activation function keeps the hyper-parameters unchanged in ANN training. For example, the initial weights of a network have to be set carefully to adapt to a constant bias on the activation function.

Parametric Activation Functions (PAFs)

Neurons in ANNs take inputs from their previous layer and feed the weighted sum of their input, $net_j = \sum_i w_{ij} x_i$, to the activation function. The transformed signal then forms the output of an artificial neuron, which can be denoted as $y_j = f(net_j)$, see Figure 6.21.

Equation 6.11 illustrates the physical interpretation of the input of an NSP function, the noisy current influx, which has the mean of m_I, and the variance of S_I^2. To express the physical parameters with the same form of the weighted summation, net, in a conventional ANN, the mean and variance of the noisy current influx can be represented with net_x and net_σ^2:

$$net\_x_j = \sum_i w_{ij}(\lambda_i \tau_{syn}), \;\; net\_\sigma_j^2 = \sum_i \left(\frac{1}{2} w_{ij}^2\right)(\lambda_i \tau_{syn}). \tag{6.15}$$

In accordance with $net_j = \sum_i w_{ij} x_i$, the input x_i of an artificial spiking neuron can be seen as:

$$x_i = \lambda_i \tau_{syn}. \tag{6.16}$$

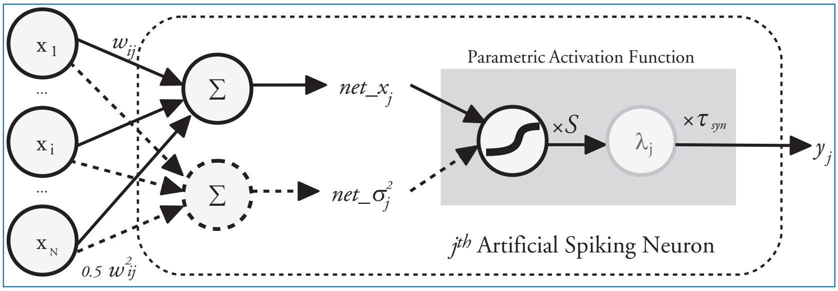


Figure 6.22. An artificial spiking neuron modelled by NSP. A spiking neuron takes firing rate λ_i as its input, which then forms the abstract numerical equivalence by multiplying the synaptic constant: $x_i = \lambda_i \tau_{syn}$. The NSP transforms the noisy current influx including both the mean (*net_x*$_j$, solid lines) and the variance *net_*σ_j^2 (dashed lines) to the abstract firing rate y_j. Finally, the output of the NSP is mapped to the physical units, firing rates (λ_j) in Hz, by multiplying S.

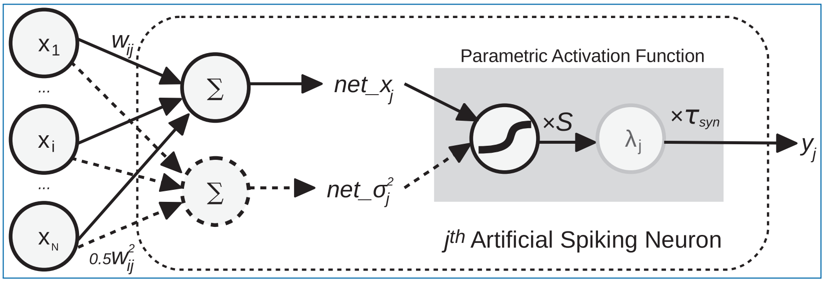


Figure 6.23. An artificial spiking neuron modelled by PAF-NSP, whose input and output are numerical values, equivalent to those of ANNs. PAF includes the scaling factors S and the synaptic time constant τ_{syn} in the combined activation function, which links the firing activity of a spiking neuron to the numerical value of ANNs.

Figure 6.22 illustrates the process that an NSP-modelled artificial spiking neuron takes the input vector **x** which is converted from the input firing rate $\boldsymbol{\lambda}$, transforms the weighted sum net_x_j and $net\_\sigma_j^2$ to the abstract output y_j and scales up y_i with the factor S to the output firing rate λ_j.

If instead of multiplying every input firing rate λ_i by τ_{syn} (left of Figure 6.22), we do it in every output firing rate after λ_j (right of Figure 6.23) and we obtain the same neuron model and structure as a typical neuron in ANNs, see Figure 6.21, that neurons take **x** as input and output abstract value y.

The only difference lies in the activation function where the artificial spiking neuron takes PAF, which is a simple linearly scaled activation function with a parameter p. The parameter is determined by the product of the scaling factor

S and the synaptic time constant τ_{syn}:

$$y = p \times f(x) = S \times \tau_{syn} \times f(x), \tag{6.17}$$

where $f(x)$ represents a typical conventional activation function, for example ReLU.

The derivative function of PAF, which is used for back propagation, is:

$$\frac{\partial y}{\partial x} = p \times f'(x) = S \times \tau_{syn} \times f'(x). \tag{6.18}$$

PAF not only allows NSP to model spiking LIF neurons on ANNs. Once the parameter p is acquired, the PAF can be generalised to other ReLU-like activation functions. Because Softplus and NSP will both converge to ReLU when the input increases, they can share the scaling factor p. Note that the calculation of noise level is not necessary for other activation functions, and thus, they only take the mean of the current influx as the input (the solid lines in Figure 6.23). So, the noise level can be set to a constant for Softplus or considered as 0 for ReLU.

It is also significant to transform numerical values of training and testing data to firing rates in the first/last layer of the SNN. To keep the firing rate in a valid range of a LIF neuron, for example, less than the maximum firing rates of $\lambda_{max} = 1/\tau_{refrac}$, we can scale the labelling data of the last layer by multiplying λ_{max}/S during training. Thus, according to PAF (Equation 6.17), the maximum firing rate of such an output neuron would be $1 \times \lambda_{max}/S \times S = \lambda_{max}$. We can certainly choose a much lower rate of λ_{max}, say 200 Hz, to keep the NSP fit to the actual LIF response activities better, since the parameters of PAF are curve-fitted to a limit working range of output firing rates. For the input layer, it is easiest to keep the original abstract values as $\boldsymbol{x}$; then, in the SNN test, we divide $\boldsymbol{x}$ by τ_{syn} to get the input firing rates of Poisson spike trains, see Equation 6.16. But, it is also flexible to linearly map the numerical values to a range of firing rates by multiplying K Hz. Then, we use $\boldsymbol{x} \times K \times \tau_{syn}$ as the new input of the training network; and $\boldsymbol{x} \times K$ as firing rates of spike trains in SNN testing.

Training Method

The simple idea of PAF presented in the previous section allows the use of common ANN training methods to obtain SNN-compatible weights. Consequently, training SNNs can be done in three simple steps:

1. Calibrate the parameters (k, b, S) for NSP which models the response firing rates of LIF neurons, thus to estimate the parameter $p = S \times \tau_{syn}$ for PAFs and to set the LIF parameter $I_{offset} = b$. Since (k, b, S) are solely dependent on the biological configurations of a LIF neuron, the same p can be shared

with different activation functions and repeatedly used for various network architectures and applications.

2. Train any feed-forward ANN with a PAF version of a ReLU-like activation function. Training compatibility allows us to choose computationally simple activation functions to increase training speed. The backpropagation algorithm updates weights using the stochastic gradient descent optimisation method to minimise the error between the labels and the predictions from the network.
3. Transfer the trained weights directly to the SNN, which should use the same LIF characteristics as those used in Step 1.

Fine Tuning

As stated above, we can train the network with any PAF version of conventional ReLU-like activation functions and then fine-tune it with PAF-NSP in the hope of improving the performance of the equivalent SNN by closely modelling the spiking neurons with NSP. Additionally, we add a small number, for example 0.01, to all the binary values of the labels on the training data. Although binary labels enlarge the disparities between the correct recognition label and the rest for better classification capability, spiking neurons seldom stay silent even with negative current influx, and thus, setting labels to 0 is not practical for training SNNs. Therefore, adding an offset relaxes the strict objective function of predicting exact labels with binary values.

There are two aspects to the fine tuning which make the ANN closer to SNNs: firstly, using the NSP activation functions causes every single neuron to run at a similar noise level as in SNNs, and thus, the weights trained by other activation functions will be tuned to fit closer to SNNs. Secondly, the output firing rate of any LIF neuron is greater than zero as long as noise exists in their synaptic input. Thus, adding a small offset on the labels directs the model to approximate practical SNNs. The result of fine tuning on a ConvNet will be demonstrated in Section 6.4.4.

6.4.4 Results

Finally, the proposed generalised SNN training method is put into practice [52, 147]. We train a 6-layer ConvNet with PAF-NSP and transfer the tuned weights to an equivalent SNN. The detailed description of the experiment is illustrated in this section. We then observe the individual neuronal activities of the trained SNN, compare the learning and recognition performance between activation functions, and estimate the power consumption of the SNN running on neuromorphic hardware.

Experiment Description

A spiking ConvNet was trained on the MNIST [140] data set, using the generalised SNN training method described above. The architecture (784-6c-6p-12c-12p-10fc) contains 28×28 input units, followed by two convolution-pooling layers with 6 and 12 convolutional kernels each, and 10 output neurons fully connected to the last pooling layer to represent the classified digit.

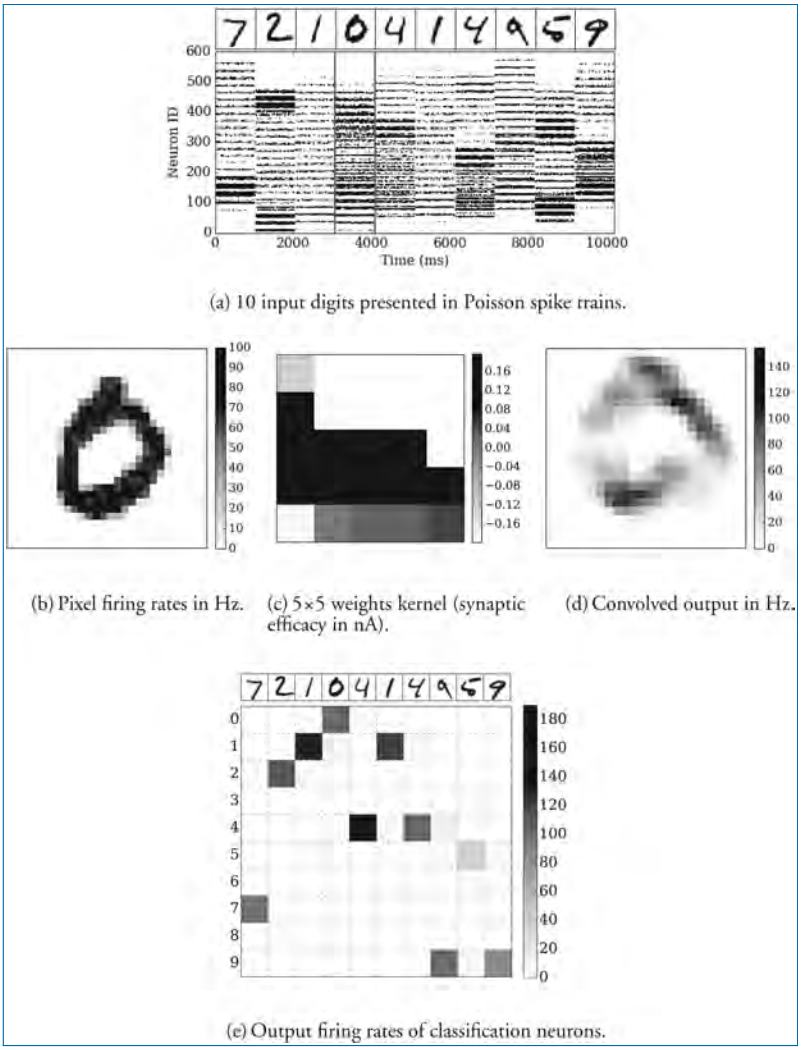
To train the ConvNet, firstly we estimated parameter p for PAFs given LIF configurations listed in Table 6.4 and $\tau_{syn} = 0.005$ s, $p = S \times \tau_{syn} = 1.085$, where $(k = 0.31, b = 0.1, S = 217)$ were calibrated using NSP. Secondly, the training employed PAFs with three core activation functions: ReLU, Softplus and NSP to compare their learning and recognition performance. The weights were updated using a decaying learning rate, 50 images per batch and 20 epochs. Finally, the trained weights were then directly transferred to the corresponding spiking ConvNets for recognition tests on the SNN simulator, NEST [77]. To validate the effect of fine tuning, we took another training epoch to train these models with PAF-NSP with data labels shifted by $+0.01$. Then, the weights were also tested on SNN simulations to compare with the ones before fine-tuning.

At the testing stage, the input images were converted to Poisson spike trains [148] and presented for 1 s each. The output neuron which fired the most indicated the classification of an input image.

Individual Neuronal Activity

To validate how well the NSP activation fits the response firing rate of LIF neurons in SNNs, we simulated one of the PAF-NSP trained ConvNets on NEST. Ten testing images were presented with spike trains whose firing rates were calculated as: $\boldsymbol{\lambda} = \boldsymbol{x}/\tau_{syn}$. The inputs were convolved with a trained 5×5 kernel, and the output firing rates of the spiking neurons were recorded, see Figure 6.24.

The recorded firing rates are compared to the predictions of these PAFs: $\lambda' = S \times f(x) = y/\tau_{syn}$, see Figure 6.25. The estimated spike counts using NSP fitted the recorded firing rate much more accurately than with ReLU or Softplus. The Euclidean distances, $\sqrt{\sum_j (\lambda'_j - \lambda_j)^2}$, between the spike counts and the firing rates predicted by NSP, ReLU and Softplus were 180.59, 349.64 and 1293.99, respectively. We manually selected a static noise level of 0.45 for Softplus, whose estimated firing rates located roughly on the top slope of the real response activity. This resulted in a longer Euclidean distance than using ReLU, since most of the input noisy currents were of relatively low noise level in this experiment. Hence, the firing rate driven by the lower noise level is closer to the ReLU curve than to Softplus.



(a) 10 input digits presented in Poisson spike trains.

(b) Pixel firing rates in Hz.

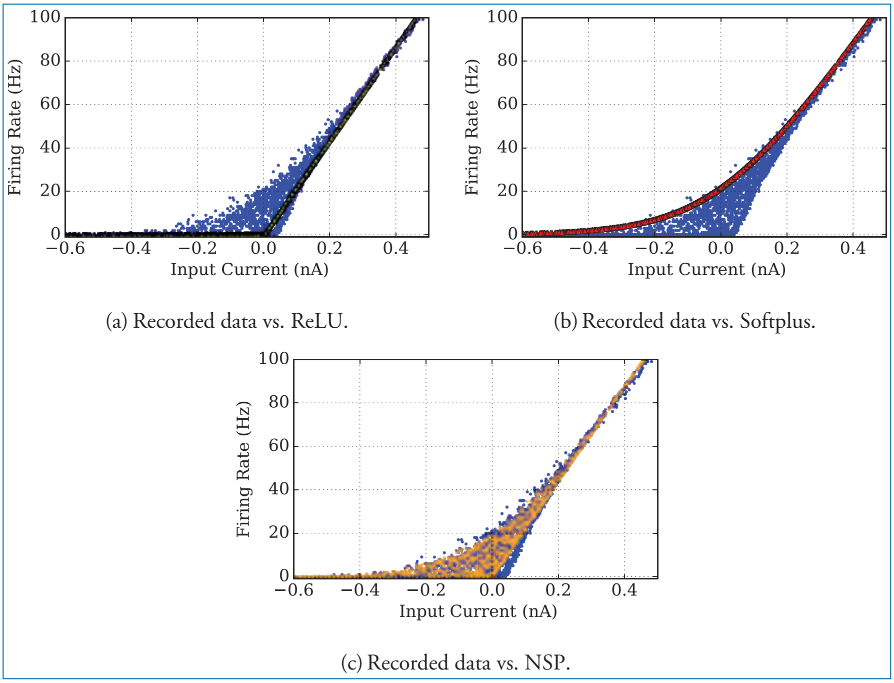
(c) 5×5 weights kernel (synaptic efficacy in nA).

(d) Convolved output in Hz.

(e) Output firing rates of classification neurons.

Figure 6.24. Images presented in spike trains convolved with a weight kernel. (a) The 28 × 28 Poisson spike trains as a raster plot, representing the 10 digits in MNIST. (b) The firing rates of all of the 784 neurons of the fourth image, digit '0', plotted as a 2D image. (c) One out of six of the trained kernels (5 × 5 size) in the first convolutional layer. (d) The spike trains plotted as the firing rates of the neurons in the convolved 2D map. (e) Output firing rates for recognising these digits.

Note that there is a visible mismatch between the actual firing rates and the model estimation in the lower right region in Figures 6.25(a), (c), where the blue dots (actual spike counts) fall below the bound of ReLU. This is consistent with the statement in Section 6.4.2 that the LIF response activities does not fit into the



(a) Recorded data vs. ReLU.

(b) Recorded data vs. Softplus.

(c) Recorded data vs. NSP.

Figure 6.25. The recorded firing rate of the convolution of the same kernel with 10 images in SNN simulation, compared to the firing rate prediction by $S \times f(x)$. NSP (c) fits to the neural response firing rate of LIF neurons more closely than ReLU (a) and Softplus (b).

NSP function when the noise level is low (approaching 0). However, the minor mismatch does not result in poor performance on classification accuracy.

Figure 6.24(e) demonstrates the output firing rates of the 10 recognition neurons when tested with the digit sequence. The SNN successfully classified the digits where the correct label neuron fired the most. We trained the network with binary labels on the output layer, and thus, the expected firing rate of correct classification was $1 \times S = 217$ Hz according to Equation 6.16. The firing rates of the recognition test fell into the valid range. This shows another advantage of NSP in that we can estimate the firing rate of an SNN by $S \times f_{NSP}(x)$ from running its equivalent ANN, instead of simulating the SNN. Moreover, we can constrain the expected firing rate of the top layer, thus preventing the SNN from exceeding its maximum firing rate, for example, 1 KHz when the time resolution of the simulation is set to 1 ms.

Learning Performance

Before looking into the recognition results, it is significant to see the learning capability of the novel activation function, NSP. We compared the training using

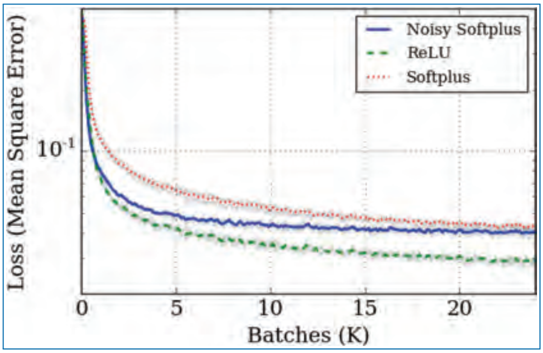


Figure 6.26. Comparisons of loss during training using NSP, ReLU and Softplus activation functions. Bold lines show the average of three training trials, and the grey colour illustrates the range between the minimum and the maximum values of the trials.

ReLU, Softplus and NSP by their loss during training averaged over three trials, see Figure 6.26. ReLU learned fastest with the lowest loss, thanks to its steepest derivative. In comparison, Softplus accumulated spontaneous 'firing rates' layer by layer and its derivative may experience gradual or even vanishing gradients during back propagation, which results in more difficult training. The recognition performance of NSP lay between these two. The loss stabilised to the same level as Softplus, because of the same problem of gradual gradients.

However, the learning stabilised fastest using NSP, which may be a result of the accurate modelling of the noise. Similar findings have shown that networks with added noise, for example, dropout [236], also improve training time. The result suggests that NSP may similarly shorten training time.

Recognition Performance

Classification accuracy: The classification errors for the tests were investigated by comparing the average classification accuracy among three trials, shown in Figure 6.27. At first, all trained models were tested on the same artificial neurons as used for training the ANNs, and these experiments were called the 'DNN' test since the network had a deep structure (6 layers). Subsequently, the trained weights were directly applied to the SNN without any transformation, and these 'SNN' experiments tested their recognition performance on the NEST simulator. From DNN to SNN, the classification accuracy declines by 0.80%, 0.79% and 3.12% on average for NSP, ReLU and Softplus.

The accuracy loss is caused by the mismatch between the activations and the practical response firing rates, see examples in Figure 6.25, and the strict binary labels for NSP and Softplus activations. Fortunately, the problem is alleviated

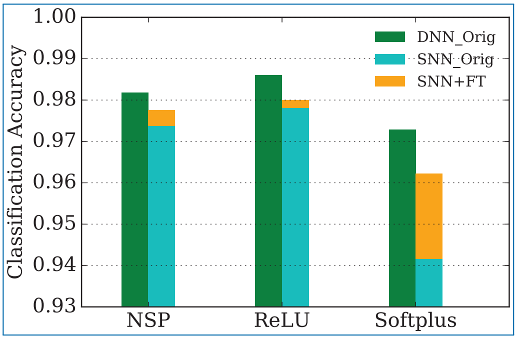


Figure 6.27. Classification accuracy. The trained weights were tested using the same activation function as training (DNN_Orig), then transferred to an SNN and tested using NEST simulation (SNN_Orig) and finally fine-tuned to be tested on an SNN (SNN_FT) again.

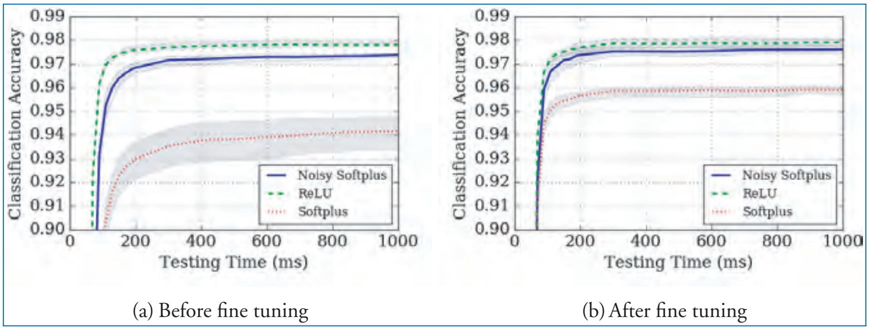
by fine-tuning, which increases the classification accuracy by 0.38%, 0.19% and 2.06% and results in total losses of 0.43%, 0.61% and 1.06%, respectively. Softplus benefited the most from fine tuning, since the larger mismatch (Figure 6.25(c)) of the response firing rate is significantly corrected. The improvement of NSP is obtained from the offset on the labels which helps the network to fit practical SNNs. As the recognition performance of ReLU is already high, there is little room for improvement. Even though the fine-tuning procedure does its job, the gain in accuracy is the smallest for this activation function.

The most efficient training in terms of both classification accuracy and algorithm complexity takes PAF-ReLU for ANN training and PAF-NSP for fine-tuning. The best classification accuracy achieved by a larger spiking ConvNet (784-16c-16p-64c-64p-10fc) was 99.07% after fine-tuning, a 0.14% drop from the ANN test (99.21%). The network reached the recognition rate of 98.7% even without fine-tuning, so we suggest making fine-tuning an optional step for training.

Comparisons in Literature: It is useful to compare with existing SNN training methods shown in Table 6.5 where we order them on their computational complexity (in descending order). The generalised training method presented here uses simple abstract activation functions, for example, PAF-ReLU; it requires no modulations of trained weights to adapt to SNNs, but uses a single *optional* additional processing of fine-tuning. The training method is well fitted to biologically plausible LIF neurons, which are supported by most neuromorphic platforms. Regarding the classification accuracy, it achieves the state-of-the-art performance of SNNs and compares favourably with all the other methods using LIF neurons. The

Table 6.5. SNN training methods comparison.

Method	Activation function	Biologically plausible	Additional processing	Weight conversion	Accuracy (%)
[122]	Siegert	**Yes**	**No**	**No**	94.94 [240]
[111]	Soft LIF	**Yes**	Noisy inputs and activations	**No**	98.37
[51]	**ReLU**	No	Dropout	Yes	**99.1**
This work	**PAF** ($p \times$ReLU)	**Yes**	**No** or fine tune	**No**	98.72 **99.07 (fine tune)**



(a) Before fine tuning

(b) After fine tuning

Figure 6.28. The classification accuracy of three trials (averaged in bold lines, grey shading shows the range between minimum to maximum) over short response times, with trained weights (a) before fine-tuning and (b) after fine-tuning.

combination of these features results in a method with exceptional performance and ease-of-use for training SNNs.

Recognition time: As this is a major concern in neuromorphic vision, the recognition performance over short response times is also estimated – see Figure 6.28. After fine-tuning, Softplus significantly reduced the mismatch since the variation over the three trials shrinks to a range similar to other experiments. Fine-tuning also improved its classification accuracy and the response latency. Notice that all of the networks trained by three different activation functions showed a very similar stabilisation curve, which means they all reached an accuracy close to their best after only 300 ms of biological time.

Power Consumption

Noisy Softplus can easily be used for energy cost estimation for SNNs. For a single neuron, the energy consumption of the synaptic events it triggers is:

$$\begin{aligned} E_j &= \lambda_j N_j T E_{syn} \\ &= \frac{y_j N_j T E_{syn}}{\tau_{syn}}, \end{aligned} \tag{6.19}$$

where λ_j is the output firing rate, N_j is the number of post-synaptic neurons it connects to, T is the testing time and E_{syn} is the energy cost for a synaptic event of some specific neuromorphic hardware, for example, about 8 nJ on SpiNNaker [242]. Thus, to estimate the whole network, we can sum up all the synaptic events of all the neurons:

$$\sum_j E_j = \frac{T E_{syn}}{\tau_{syn}} \sum_j y_j N_j. \tag{6.20}$$

Thus, it may cost SpiNNaker 0.064 W, 192 J running for 3,000 s with synaptic events of $8 \times 10^6/s$ to classify 10,000 images (300 ms each) with an accuracy of 98.02%. The best performance reported using the larger network may cost SpiNNaker 0.43 W operating synaptic event rate at 5.34×10^7 Hz, consuming 4271.6 J to classify all the images for 1 s each.

6.4.5 Summary

We presented a generalised off-line SNN training method to tackle the research problem of equipping SNNs with equivalent cognitive capability to ANNs. This training procedure consists of three simple stages: first, estimate parameters for PAF using NSP; second, use a PAF version of conventional activation functions for ANN training; third, the trained weights can be directly transferred to the SNN without any further transformation.

Regarding the generalisation, the training not only uses popular activation functions in ANNs, for example, ReLU, but also targets standard LIF neurons which are widely used on neuromorphic hardware. Therefore, the proposed method greatly simplifies the training of AI applications for neuromorphic hardware, thereby paving the way to energy-efficient AI on brain-like computers: from neuromorphic robots to clusters. Moreover, it lowers the barrier for AI engineers to access neuromorphic hardware without the need to understand SNNs or the hardware. Furthermore, this method incurs the least computational complexity while performing the most effectively among existing algorithms. In terms of classification/recognition accuracy, the performance of ANN-trained SNNs is nearly equivalent to ANNs, and the performance loss can be partially offset by fine-tuning. The best classification accuracy of 99.07% using LIF neurons in a PyNN simulation outperforms state-of-the-art SNN models of LIF neurons and is equivalent to the best result achieved using IF neurons. Another important feature of accurately modelling LIF neurons in ANNs is the acquisition of spiking neuron firing rates. These will aid deployment of SNNs in neuromorphic hardware by providing power and communication estimates, enabling better use or customisation of the hardware platforms.

DOI: 10.1561/9781680836530.ch7

Chapter 7

Learning in Neural Networks

By Petruț Bogdan, Garibaldi Pineda García, Michael Hopkins, Edward Jones, James Knight and Adam Perrett

> *If you don't sleep the very first night after learning, you lose the chance to consolidate those memories, even if you get lots of 'catch-up' sleep thereafter. In terms of memory, then, sleep is not like the bank. You cannot accumulate a debt and hope to pay it off at a later point in time. Sleep for memory consolidation is an all-or-nothing event.*
>
> — Matthew Walker

A very important set of open questions in Neuroscience are related to learning, from how addiction rewires our brains to how you can remember where you parked your car or left your bike this morning, but can't remember why you entered the kitchen. Neural memories, whether artificial or biological, seem to operate over multiple time scales. Very short-term memories are fast, but limited so get overwritten often. Long-term memories can stick around a lifetime, but they take a good night's sleep to consolidate. Whether through sleep spindles or one-shot learning, brains utilise synaptic plasticity to store these patterns of activity that we call memories, concepts or motor actions.

This chapter is concerned with the motivation, design and implementation behind mimicking biological learning rules with a focus on, you guessed it, SpiNNaker. It starts by presenting Spike-timing-dependent plasticity (STDP) operating in an unsupervised fashion based on relative spike times of the pre- and post-synaptic neurons or based on the sub-threshold membrane potential. This is followed by a model of STDP modulated by the presence of an additional signal and operating on eligibility traces. Longer-term mechanisms in the form of structural plasticity, involving the rewiring of connections between the neurons, and (very long-term) neuroevolution close out the chapter.

7.1 Sizing Up the (Biological) Competition

A quick detour: Let's talk brains.

The combination of the size of the human brain, in terms of the number of neurons ($\approx 8.8 \times 10^{10}$ [7]) and synapses ($\approx 1.5 \times 10^{14}$ [180, 187]) and the different possible scales of exploration, from gene expression to behaviour, makes the task of mapping the human brain intractable. As a result, our approach focuses on the simulation of brain regions, relying on sparse, but strategic data. Previous attempts have usually been data driven, attempting to replicate at some level the operation of brain cells [100], regions [154] or processes [29, 118], or concept driven, wherein the modellers are interested in harnessing the computational power of whatever it is they are modelling [11, 54, 124]. Further, while the brain is currently the gold standard for computational power, function and flexibility, and a lot of effort is being invested in engineering systems with similar performance, it might be worth remembering that the brain is the result of millions of years of evolution. In its current form, it is a tangled mess that various institutions and large-scale projects (e.g. Amunts *et al.* [4]) are trying to mine for strategic data, but it is not all gold mine.

Some 'features' of the brain are short-cuts, heuristics to deal with vast amounts of sensory information, and so they are fallible [28].

Designing neural circuits that could overcome biological limitations might be more important than being content with replicating the brain's capabilities. Of course, since some computations are not tractable even under the assumption of clairvoyance (e.g. see the scheduling chapter in Christian and Griffiths [36]), heuristics need to be employed and thus accuracy sacrificed.

Regardless of whether we beat brains in terms of performance (spoiler: we haven't yet), the rest of the chapter should be an interesting exploration of learning techniques in SNNs.

7.2 Spike-Timing-Dependent Plasticity

In this section, we will consider only the changing of the strength of *existing* connections and, in this context, Hebb's postulate indicates that connections between neurons which persistently fire at the same time will be strengthened. Neurons which persistently fire at the same time are likely to do so because they respond to similar or related stimuli.

Bliss and Lømo [20] provided the first evidence to support this hypothesis by measuring how – if two connected neurons are stimulated simultaneously – the synaptic connections between them are strengthened. In networks of rate-based neurons, this behaviour has been modelled using rules such as the Bienenstock, Cooper, Munroe (BCM) rule [18] and Oja's rule [184]. However, the focus of this section is on SNNs, and in such networks, the timings of individual spikes have been shown to encode both temporal and spatial information. Therefore, in this section, we focus on STDP – a form of synaptic plasticity capable of learning such timings.

In Section 7.2.1, we outline some of the experimental evidence supporting STDP and discuss how STDP can be modelled in networks of spiking neurons. Then, in Section 7.2.2, we discuss how STDP has previously been implemented on SpiNNaker and other distributed systems.

We have developed a new SpiNNaker STDP implementation which has both lower algorithmic complexity than prior approaches and employs new low-level optimisations to exploit the ARM instruction set better. Improving the performance of previous SpiNNaker STDP implementations is an important aspect of this work. It is analysed and presented in detail by Knight [129]. Finally, in Section 7.2.3, we discuss this implementation in depth. This new implementation is now a key component of the SpiNNaker software developed as part of the HBP which aims to provide a common platform for running PyNN simulations on SpiNNaker, BrainScaleS and HPC platforms.

7.2.1 Experimental Evidence for Spike-Timing-Dependent Plasticity

Levy and Steward [142] showed that if the experiment performed by Bliss and Lømo [20] was repeated with a delay between the stimulation of two neurons, then the magnitude of the increase in weight could be reduced or even reversed. Subsequently, Bi and Poo [16] measured the changes in synaptic efficacy induced in the synapses of hippocampal neurons by pairs of pre- and post-synaptic spikes with different relative timings. The relationship between the magnitude of these changes and the relative timing of the pre- and post-synaptic spikes is known as STDP, and the data recorded by Bi and Poo suggest that it reinforces causality between the firing of the pre- and post-synaptic neurons. When a pre-synaptic spike arrives before a post-synaptic spike is emitted, the synapse is *potentiated* (strengthened). However, if a pre-synaptic spike arrives after a post-synaptic spike has been emitted, the synapse is *depressed* (weakened). Furthermore, the data recorded by Bi and Poo

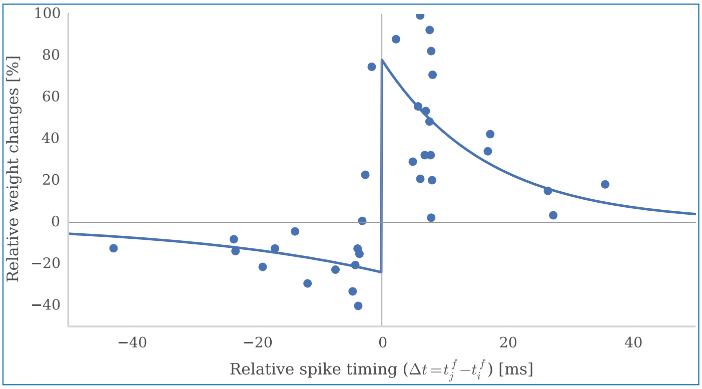


Figure 7.1. Excitatory STDP curve. Each dot represents the relative change in synaptic efficacy after 60 pairs of spikes. After Bi and Poo [16].

suggest that the magnitude of changes in synaptic efficacy (Δw_{ij}) is related to the relative spike timings with the following exponential functions (Figure 7.1):

$$\Delta w_{ij} = \begin{cases} F_{+}(w_{ij}) \exp\left(-\frac{\Delta t}{\tau_{+}}\right) & \text{if } \Delta t > 0 \\ F_{-}(w_{ij}) \exp\left(\frac{\Delta t}{\tau_{-}}\right) & \text{if } \Delta t \leq 0 \end{cases} \tag{7.1}$$

Where $\Delta t = t_j - t_i$ represents the relative timing of pre- and post-synaptic spikes, $\tau_{\pm}$ defines the time constant of the exponentials and the $F_{\pm}$ functions define how the magnitude of the change in weight depends on the current synaptic efficacy.

Bi and Poo [16] measured how Δw_{ij} depended on the previous value of w_{ij}. In Figure 7.2, we redraw their data on a double-logarithmic scale and fit straight lines to the potentiation and depression components (as suggested by Morrison *et al.* [170]). The nature of the $F_{\pm}$ functions is indicated by the gradient of each line. Since the trend line through the depression data has a gradient of -1, it would suggest that F_{-} is linearly proportional to the weight. However the nature of F_{+} is less clear as the trend line through the potentiation data has a gradient of 0.4. Gütig *et al.* [86] and Morrison *et al.* [170] proposed using the power law function $F_{+} \propto {w_{ij}}^{\mu}$ to represent the magnitude of the change in weight in response to potentiation; $\mu = 0$ makes the weight update independent of the previous weight; and $\mu = 1$ makes the update linearly proportional to the previous weight. With $\mu = 0.4$, the linear fit to the potentiation data recorded by Bi and Poo can be obtained.

Another common means of describing STDP rules is by using ‘trace variables’ [170, 171, 210] which get updated when pre- and post-synaptic spikes occur and represent the combined effects of the preceding pre- and post-synaptic spikes. For example, the interactions between individual pairs of spikes described by

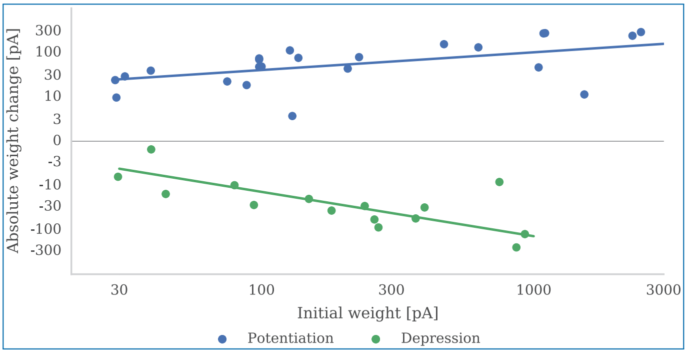


Figure 7.2. Absolute change in synaptic efficacy after 60 spike pairs. Potentiation is induced by spike pairs where the pre-synaptic spike precedes the post-synaptic spike by 2.3 ms to 8.3 ms. Depression is induced by spike pairs in which the post-synaptic spike precedes the pre-synaptic spike by 3.4 ms to 23.6 ms. The upper blue line is a linear fit to the potentiation data with slope: 0.4. The lower green line is a linear fit to the depression data with slope: −1. After Morrison *et al.* [170].

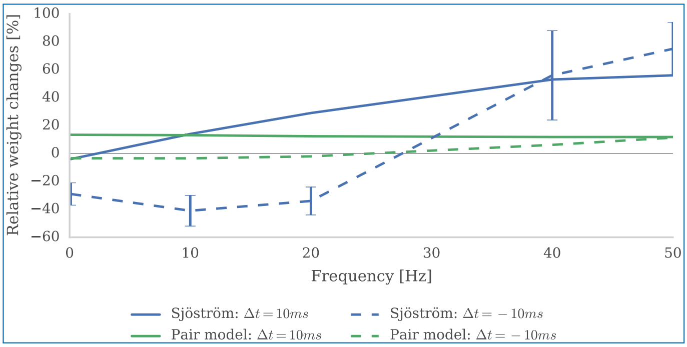


Figure 7.3. Fitting a pair-based STDP model with $\tau_+ = 16.8$ ms and $\tau_- = 33.7$ ms to data from Sjöström *et al.* [229] by minimising the mean squared error fails to reproduce frequency effects. Blue lines and data points redrawn from Sjöström *et al.* and the green lines show the best fit obtained by the pair-based STDP model. After Pfister [192].

Equation 7.1 can alternatively be modelled based on pre- (s_i) and post-synaptic (s_j) trace variables:

$$\frac{ds_i}{dt} = -\frac{s_i}{\tau_+} + \sum_{t_i^f} \delta(t - t_i^f) \tag{7.2}$$

$$\frac{ds_j}{dt} = -\frac{s_j}{\tau_-} + \sum_{t_j^f} \delta(t - t_j^f) \tag{7.3}$$

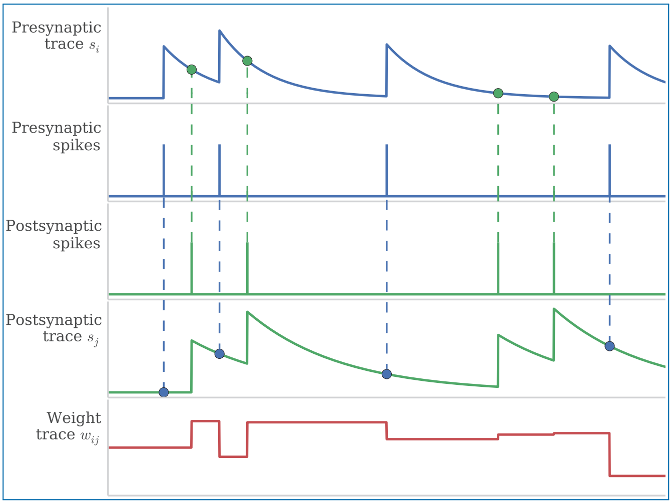


Figure 7.4. Calculation of weight updates using pair-based STDP traces. Pre- and post-synaptic traces reflect the activity of pre- and post-synaptic spike trains. Potentiation is calculated at each post-synaptic spike time by sampling the pre-synaptic trace (green circle) to obtain a measure of recent pre-synaptic activity. Depression is calculated at each pre-synaptic spike time by sampling the post-synaptic trace (blue circle) to obtain a measure of recent post-synaptic activity. Weight dependence is additive. After Morrison *et al.* [171].

Pre- and post-synaptic spikes occurring at t_i^f and t_j^f, respectively, are modelled using Dirac delta functions (δ) and, as the top 4 panels of Figure 7.4 show, the trace variables represent a low-pass filtered version of these spikes. These dynamics can be thought of as representing chemical processes. For example s_i can be viewed as a model of glutamate neurotransmitters which, having crossed the synaptic cleft from the pre-synaptic neuron, bind to receptors on the post-synaptic neuron and are reabsorbed with a time constant of τ_+. Building on this work, Section 7.4 presents an implementation of neuromodulated STDP simulated on SpiNNaker.

As the dashed blue lines in Figure 7.4 illustrate, when a pre-synaptic spike occurs at time t_i^f, the s_j trace can be sampled to obtain the combined depression caused by the pairs made between this pre-synaptic spike and all preceding post-synaptic spikes. Similarly, as the dashed green lines in Figure 7.4 illustrate, when a post-synaptic spike occurs at time t_j^f, the s_i trace can be sampled, leading to the following equations for calculating depression (Δw_{ij}^-) and potentiation (Δw_{ij}^+):

$$\Delta w_{ij}^-(t_i^f) = F_-(w_{ij}) s_j(t_i^f) \tag{7.4}$$

$$\Delta w_{ij}^+(t_j^f) = F_+(w_{ij}) s_i(t_j^f) \tag{7.5}$$

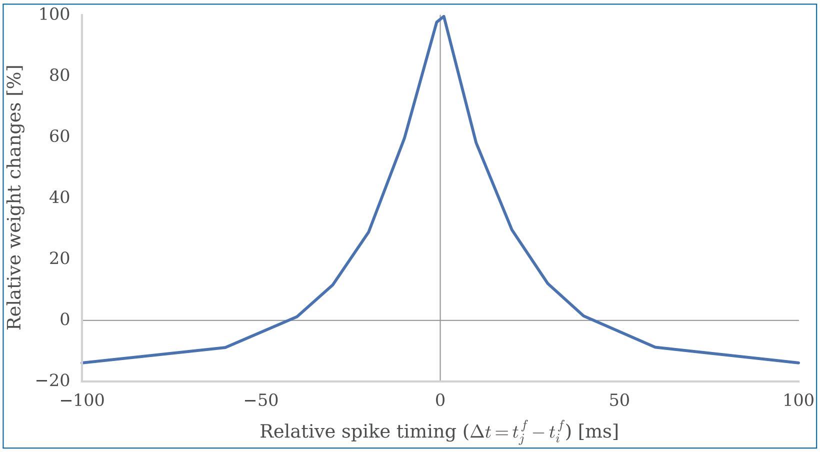


Figure 7.5. Inhibitory STDP curve. The relative change in synaptic efficacy after 60 pairs of spikes. After Vogels *et al.* [261].

Bi and Poo [16] recorded the data plotted in Figures 7.1 and 7.2 from rat hippocampal neurons, but subsequent studies have revealed similar relationships – albeit with different time constants and polarities – in other brain areas [210]. Specifically, in the neocortex, excitatory synapses appear to exhibit STDP with similar asymmetrical kernels to hippocampal neurons, whereas inhibitory synapses have a symmetrical kernel similar to that shown in Figure 7.5.

While rules that consider pairs of spikes provide a good fit for the data measured by Bi and Poo, they cannot account for effects seen in more recent experimental data. Sjöström *et al.* [229] stimulated cortical neurons with pairs of pre- and postsynaptic spikes separated by a constant 10 ms but with between 20 ms and 10 s separating the pairs. When the time between the pairs approaches the time constants defining the temporal range of the pair-based STDP rule, spikes from neighbouring pairs begin to interact. As shown in Figure 7.3, this interaction then cancels out the potentiation or depression that the original pair should have elicited.

Several extensions to the STDP rule have been proposed which take into account the effect of multiple preceding spikes including the 'triplet rule' proposed by Pfister [192]. In this rule, the effect of earlier spikes is modelled using a second set of traces (s_i^2 and s_j^2) with longer time constants τ_x and τ_y:

$$\frac{ds_i^2}{dt} = -\frac{s_i^2}{\tau_x} + \sum_{t_i^f} \delta(t - t_i^f) \tag{7.6}$$

$$\frac{ds_j^2}{dt} = -\frac{s_j^2}{\tau_y} + \sum_{t_j^f} \delta(t - t_j^f) \tag{7.7}$$

To incorporate the effect of these traces into the weight updates, Pfister also extended Equations 7.4 and 7.5:

$$\Delta w_{ij}^{-}(t_i^f) = s_j(t_i^f)(A_2^- + A_3^- s_i^2(t_i^f - \epsilon)) \tag{7.8}$$

$$\Delta w_{ij}^{+}(t_j^f) = s_i(t_j^f)(A_2^+ + A_3^+ s_j^2(t_j^f - \epsilon)) \tag{7.9}$$

Where ϵ is a small positive constant used to ensure that the second set of s^2 traces is sampled just *before* the spike occurs at t_i^f or t_j^f. This rule has an explicitly additive weight dependence with the relative effect of the four traces controlled by the four free parameters A_2^+, A_2^-, A_3^+ and A_3^-. Pfister fitted these free parameters to the data obtained by Sjöström *et al.* [229] and, as shown in Figure 7.6, demonstrated that the rule can accurately reproduce the frequency effect measured by Sjöström *et al.*

The trace-based models we have discussed so far assume that all preceding spikes can affect the magnitude of STDP weight updates. However experimental data [229] suggest that this might not be the case and that basing pair-based STDP weight updates on only the most recent spike can improve the fit of these models to experimental data. This 'nearest-neighbour' spike interaction scheme can be implemented in a trace-based model by resetting the appropriate trace to 1 when a spike occurs rather than by incrementing it by 1. Pfister also investigated the effect of different spike interaction schemes on their triplet rule but found it had

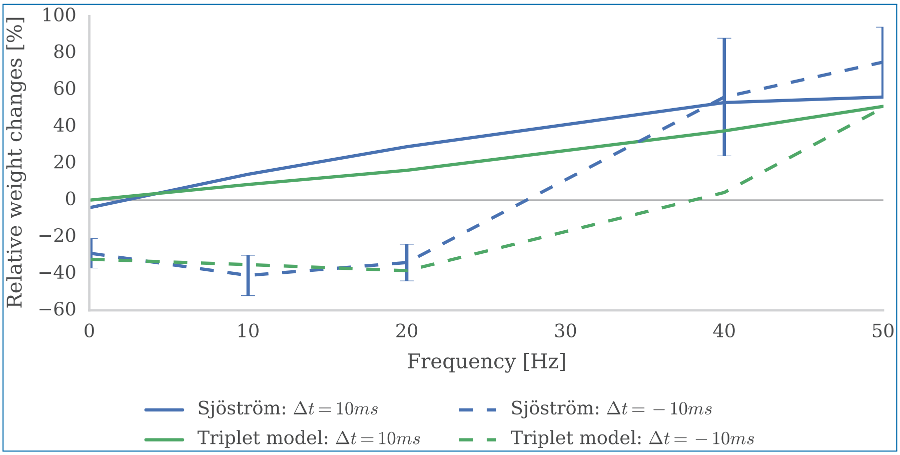


Figure 7.6. Fitting triplet STDP model with $\tau_+ = 16.8$ ms, $\tau_- = 33.7$ ms, $\tau_x = 101$ ms and $\tau_y = 125$ ms to the data recorded by Sjöström *et al.* [229] by minimising mean squared error effectively reproduces frequency effects. Blue lines and data points (with errors) redrawn from Sjöström *et al.* and the green lines show the best fit obtained by the triplet STDP model. After Pfister [192].

no significant effect on its fit to the data recorded by Sjöström *et al.* This suggests that alternative spike-pairing schemes may simply be another means of overcoming some of the limitations of pair-based STDP models.

7.2.2 Related Work

Implementing the STDP rules discussed in Section 7.2.1 in a naïve manner is relatively trivial. However, implementing them in a manner suitable for large scale simulation on a distributed system such as SpiNNaker is more difficult.

The efficient access to synaptic weights required by the event-driven synaptic processing algorithm is facilitated by storing the synaptic matrices in a row-major format. Consequently, when a pre-synaptic spike arrives, weight updates (Equation 7.4) can be evaluated on a row which is *contiguous* in memory. However, when a post-synaptic spike is emitted, weight updates (Equation 7.3) must be evaluated on a *non-contiguous* synaptic matrix *column*. Accessing synaptic matrix columns is problematic at the hardware level as the SpiNNaker DMA controller can fetch only contiguous blocks of data. Moreover, because connectivity in the neocortex is relatively sparse, synaptic matrices are represented using a compressed sparse row structure which does *not* provide efficient access to matrix columns. To remove the need for column accesses, all of the STDP implementations presented in this section defer outgoing post-synaptic spikes to allow post-synaptic weight updates to be deferred until the next *pre-synaptic* spike occurs.

An additional problem regards synaptic delays. Delays are simulated on SpiNNaker by inserting synaptic weights into an input ring buffer. However the relative timing of pre- and post-synaptic spikes, and therefore the outcome of STDP, depends on how much of this total delay occurs in the pre-synaptic axon and how much in the post-synaptic dendritic tree. As shown in Figure 7.7, pre-synaptic spikes travelling down the pre-synaptic axon to the synapse incur an 'axonal delay' and post-synaptic spikes propagating back through the post-synaptic dendritic tree to the synapse incur a 'dendritic delay'.

The approaches discussed in this section differ largely in the algorithms and data structures they use to perform the deferral of post-synaptic spikes and to

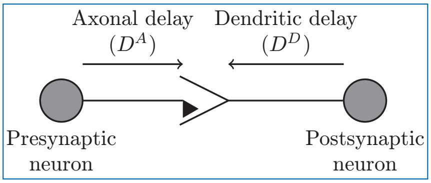


Figure 7.7. The dendritic and axonal components of synaptic delay. After Morrison *et al.* [171].

incorporate synaptic delays into the STDP processing. Jin *et al.* [121] were the first to implement STDP on SpiNNaker. They assumed that the whole synaptic delay was axonal implying that, as pre-synaptic spikes reach the synapse before this axonal delay has been applied, they too must be buffered. Jin *et al.* used a compact data structure for buffering both pre-synaptic and post-synaptic spikes containing the time at which the neuron last spiked and a bit field, the bits of which indicate previous spikes in a fixed window of time. Consequently, only a small amount of DTCM is required to store the deferred spikes associated with each post-synaptic neuron. However, because this approach does not use the trace-based STDP model (Section 7.2.1), the effect of *all possible pairs* of pre- and post-synaptic spikes must be calculated separately using Equation 7.1. Additionally, the bit field based recording of history – while compact – represents only a fixed window of time meaning that only a very small number of spikes from slow firing neurons can ever be processed.

Diehl and Cook [50] developed the first trace-based STDP implementation for SpiNNaker. To store the pre- and post-synaptic traces, they extended each synapse in the synaptic row to contain the values of the traces at the time of the last update. They allowed synapses to have arbitrary axonal and dendritic delays meaning that, like Jin *et al.*, they stored a history of both pre- and post-synaptic spikes. However, rather than using a bit field to store this, they used a fixed-size circular buffer to store the spike times. This data structure is not only faster to iterate over than a bit field but also holds a constant number of spikes, regardless of the firing rates of the pre- and post-synaptic neurons. However, these buffers can still overflow, leading to spikes not being processed if the pre- and post-synaptic firing rates are too different. For example, consider a buffer with space for ten entries being used to defer the spikes from a post-synaptic neuron firing at 10 Hz. If one of the neuron's input synapses only receives spikes (and is thus updated) at 0.1 Hz, there is insufficient buffer space for all $100 = \frac{10\,\text{Hz}}{0.1\,\text{Hz}}$ of the post-synaptic spikes that occur between the updates. Using these spike histories, Diehl and Cook developed an algorithm to perform trace-based STDP updates whenever the synaptic matrix row associated with an incoming spike packet is retrieved from the SDRAM. The algorithm loops through these synapses and, for each one, iterates through the buffered pre- and post-synaptic spikes in the order that they occurred since the last update (taking into account the dendritic and axonal delays). The effect of each buffered spike is then applied to the synaptic weight (using Equation 7.4 for pre-synaptic spikes and Equation 7.5 for post-synaptic spike) and the appropriate trace updated (using Equation 7.2 for pre-synaptic spikes and Equation 7.3 for post-synaptic spike). Diehl and Cook measured the performance of their approach using a benchmark network of 50 LIF neurons stimulated by a large number of 250 Hz Poisson spike sources connected with 20% sparsity. Using this network, they showed that their approach could process 500×10^3 incoming synaptic events per second compared to the 50×10^3 achievable using the approach developed by Jin *et al.*

There are many similarities between simulating large spiking neural networks on SpiNNaker and on other distributed computer systems – including the two problems identified at the beginning of this section. In the distributed computing space, Morrison *et al.* [170] addressed these in ways highly relevant to a SpiNNaker implementation. Although the nodes of the distributed systems they targeted do not have to access synaptic matrix rows using a DMA controller, accessing non-contiguous memory is also costly on architectures with hardware caches. Therefore, post-synaptic weight updates still need to be deferred until a pre-synaptic spike. As each node has significantly more memory, Morrison *et al.* use a dynamic data structure to guarantee that all deferred post-synaptic spikes get processed.

Morrison *et al.* simplify the model of synaptic delay by supporting only configurations where the axonal delay is shorter than the dendritic delay. This simplification allows pre-synaptic spikes to be processed immediately as it guarantees that post-synaptic spikes emitted before the axonal delay has elapsed will never 'overtake', and thus need to be processed before, the pre-synaptic spike.

This simplification means that only the time of the last pre-synaptic spike and the value of the pre-synaptic trace at that time need to be stored with each synaptic matrix row. Based on this simplification, the algorithm developed by Morrison *et al.* loops through each synapse in the row and, for each one, loops through the buffered post-synaptic spikes. The effect of each buffered spike is then applied to the synaptic weight (using Equation 7.5). After all of the post-synaptic spikes have been processed, the effect of the pre-synaptic spike that instigated the update is applied to the synaptic weight (using Equation 7.4). Once all of the synapses in the row have been processed, the pre-synaptic trace is updated (using Equation 7.2).

To assess the relative algorithmic complexity of the approaches presented in this section, we can consider the situation where an STDP synapse is updated based on N^{pre} pre-synaptic and N^{post} post-synaptic spikes. In the approach developed by Jin *et al.* [121], each pair of spikes is processed individually and the complexity is $O(N^{pre}N^{post})$. However, by using a trace-based approach, Diehl and Cook [50] reduced this complexity to $O(N^{pre} + N^{post})$ and Morrison *et al.* [170] further reduced this to $O(N^{post})$ by removing the need to buffer pre-synaptic spikes.

7.2.3 Implementation

The best performing SpiNNaker STDP implementation presented in the previous section was that developed by Diehl and Cook [50]. Their benchmark indicated that, using their implementation, a SpiNNaker core could process up to 500×10^3 incoming synaptic events per second, compared with 5×10^6 events for non-plastic synapses. As this corresponds to a significant reduction in the size of model a given SpiNNaker machine can simulate, improving the performance of STDP is an important part of enabling large-scale neocortical simulation on SpiNNaker.

We have developed a new SpiNNaker STDP implementation based on the algorithm developed by Morrison *et al.* [170] which has lower algorithmic complexity than previous SpiNNaker implementations and employs new low-level optimisations to exploit the ARM instruction set better. In this section, we present the details of this new implementation and demonstrate how it addresses the previously identified problems with distributed simulation of STDP.

As discussed in Section 7.2.2, Diehl and Cook used a fixed-sized data structure with space for 10 events to store the post-synaptic history. Using this system, if more than 10 post-synaptic spikes backpropagate to a synapse between updates, some will be lost. Based on the distributions of cortical neuron firing rates in rats and macaque monkeys presented by Buzsáki and Mizuseki [30], we can derive the distribution of firing rate ratios between pairs of neurons shown in Figure 7.8. Based on these ratio distributions, we can determine that a buffer with 10 entries will be sufficient to handle the activity at 90% of synapses. However to prevent post-synaptic spikes being lost when the pre- and post-synaptic neurons have very different firing rates, we developed an additional mechanism called ‘flushing’ to force the processing of these spikes. This mechanism uses one bit in the 32-bit ID associated with each neuron to signify whether the neuron is emitting a ‘flush’ or an actual spike event. To determine when these events should be sent, each neuron tracks its Inter-spike interval (ISI) and, if this is *bufferSize* times longer than the ISI corresponding to the maximum firing rate of the network, a flush event is emitted.

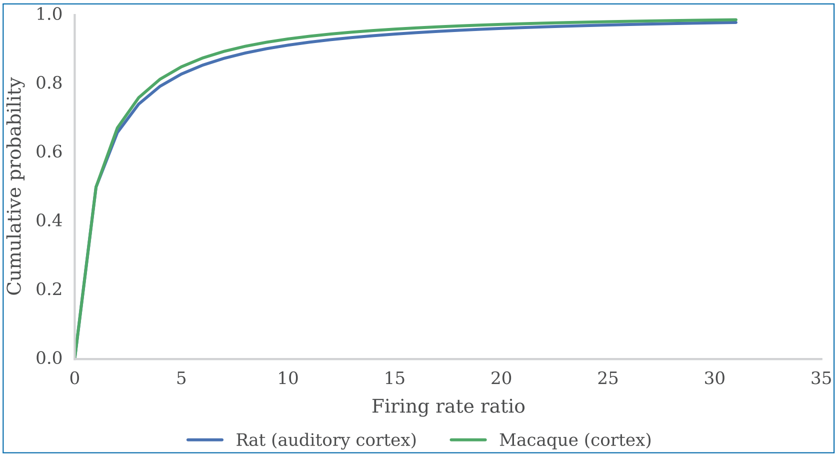


Figure 7.8. Ratio distributions of cortical firing rates. Calculated from firing rate distributions presented by Buzsáki and Mizuseki [30].

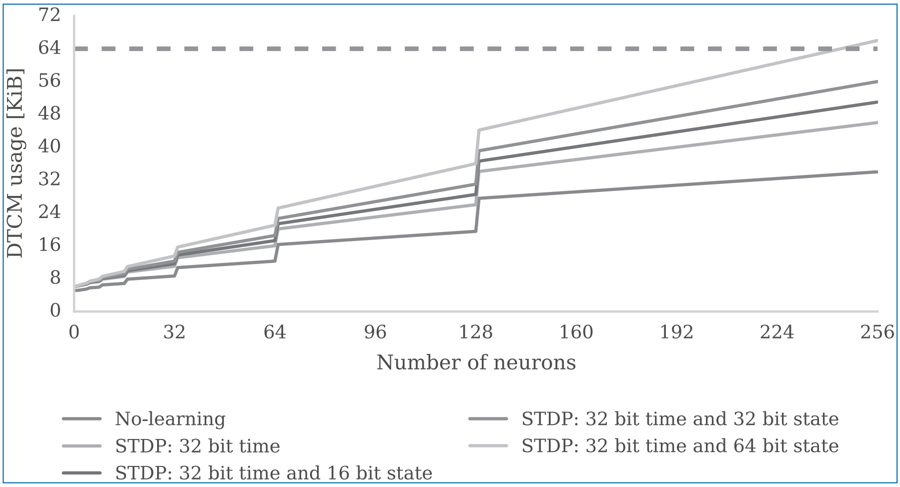


Figure 7.9. DTCM memory usage of STDP event storage schemes. The memory usage of other components is based on the current SpiNNaker tools. All trace-based schemes assume times are stored in a 32-bit format and traces in a 16-bit format, with two look-up tables with 256 16-bit entries providing exponential decay. The dashed horizontal line shows the maximum available DTCM.

Figure 7.9 shows the local memory requirements of post-synaptic history structures with capacity for 10 entries of different sizes. To implement STDP rules such as the triplet rule discussed in Section 7.2.1, each entry needs to be large enough to hold not only a spike time but also two trace values. Figure 7.9 suggests that, to avoid further reductions in the number of neurons that each SpiNNaker core can simulate, each of these traces should be represented as a 16-bit value. Using 16-bit trace entries has an additional advantage as the ARM 968 CPU used by SpiNNaker includes single-cycle instructions for multiply and multiply-accumulate operations on signed 16-bit integers [62]. These instructions allow additive weight updates such as $w_{ij} \leftarrow w_{ij} + s_j \exp\left(\frac{-dt}{tau}\right)$ to be performed using a single *SMLAxy* instruction and, when implementing rules such as the triplet rule that require two traces, they provide an efficient means of operating on pairs of 16-bit traces stored within a 32-bit field.

The range of fixed-point numeric representations is static. Thus, the optimal representation for storing traces must be chosen ahead of time based on the maximum expected value. We can calculate this by considering the value of a trace x with time constant τ after n spikes emitted at f Hz:

$$x(n) = \sum_{i=0}^{n} e^{-\frac{i}{\tau f}} \tag{7.10}$$

This can be rearranged into the form of a geometric sum:

$$x(n) = \sum_{i=0}^{n} (e^{-\frac{1}{\tau f max}})^i \tag{7.11}$$

Which has the value:

$$x(n) = \frac{1 - (e^{-\frac{1}{\tau f}})^n}{1 - e^{-\frac{1}{\tau f}}} \tag{7.12}$$

Since $|e^{-\frac{1}{\tau f}}| < 1$, as $n \to \infty$ this converges to:

$$x_{max} = \frac{1}{1 - e^{-\frac{1}{\tau f}}} \tag{7.13}$$

The sustained firing rate of most neurons is constrained by the time that ion pumps take to return the neuron's membrane potential to its resting potential. This generally limits a neuron's maximum firing rate to around 100 Hz but, as Gittis *et al.* [79] discuss, there are mechanisms that can overcome this limit. For example, vestibular nucleus neurons can maintain sustained firing rates of around 300 Hz. Figure 7.10 shows that – based on this worst-case maximum firing rate – 4 integer bits are required to store traces with time constants in the range fitted to the data recorded by Bi and Poo [16]. Therefore, a 16-bit fixed-point numeric representation with 4 integer, 11 fractional bits and a sign bit is the optimal choice for representing the traces required for pair-based STDP.

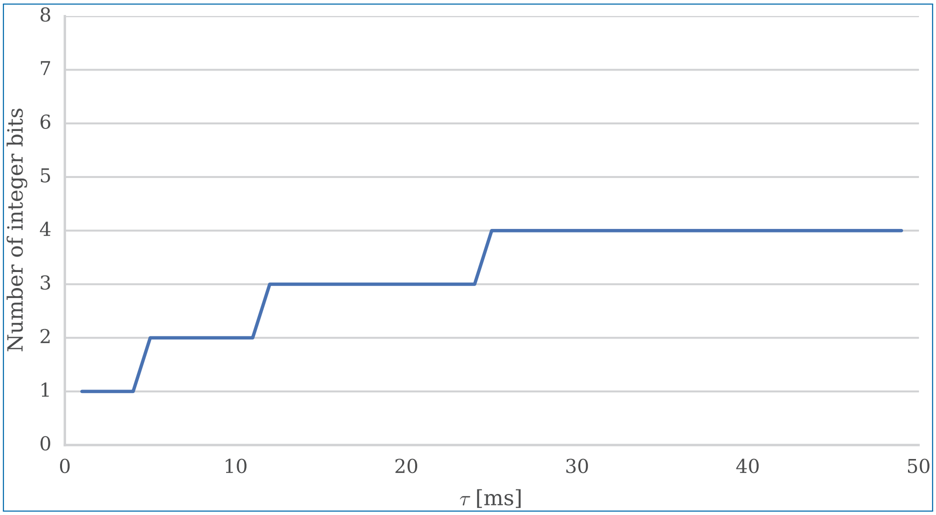


Figure 7.10. Number of integer bits required to represent traces of a 300 Hz spike train with different time constants.

In the PyNN programming interface, STDP learning rules are defined in terms of three components:

The timing dependence: Defines how the relative timing of the pre- and post-synaptic spikes affects the magnitude of the weight update.

The weight dependence: Defines how the current synaptic weight affects the magnitude of the weight update (the F_+ and F_- functions discussed in Section 7.2.1).

The voltage dependence: Defines how the membrane voltage of the post-synaptic neuron affects the magnitude of the weight update.

Adding a voltage dependence to the type of event-based STDP implementation discussed here presents several challenges beyond the scope of this section. However, one such voltage-dependent implementation is described in Section 7.3.

In this section, we implement only the timing and weight dependencies supported by PyNN. So as to allow users of the HBP software not only to select from the weight dependencies specified by PyNN but also to implement their own easily, this implementation defines simple interfaces which timing and weight dependencies must implement. Timing dependencies must define the correct types for the pre- and post-synaptic states (s_i and s_j, respectively), functions to update pre- and post-synaptic trace entries based on the time of a new spikes (*updatePreTrace* and *updatePostTrace*, respectively) and functions to apply the effect of deferred pre- and post-synaptic spikes to a synaptic weight (*applyPreSpike* and *applyPostSpike*, respectively). Algorithm 1 shows an implementation of the functions required to implement pair-based STDP using this interface. The *updatePreTrace* adds the effect of a new pre-synaptic spike at time t to the pre-synaptic trace by decaying the value of s_i calculated at the time of the last spike ($t^{\text{lastSpike}}$) and adding 1 to represent the effect of the new spike (the closed-form solution to Equation 7.2 between two t_i^f s). Similarly, the *applyPreSpike* function samples the post-synaptic trace by decaying the value of s_j calculated at the time of the last post-synaptic spike (t_j) (the $s_j(t_i^f)$ term of Equation 7.4).

To decouple the timing and weight dependencies, the *applyPreSpike* and *applyPostSpike* functions in the timing dependence call the *applyDepression* and *applyPotentiation* functions provided by the weight dependence rather than directly manipulating w_{ij} themselves. Algorithm 2 shows an implementation of *applyDepression* which performs an additive weight update.

Algorithm 3 is the result of combining the simplified delay model proposed by Morrison *et al.* [170] with the flushing mechanism and the interfaces for timing and weight dependencies discussed in this section. The algorithm begins by looping through each post-synaptic neuron (j) in the row and retrieving a list of the

Algorithm 1 Pair-based STDP timing-dependence implementation. Equivalent *updatePostTrace* and *applyPostTrace* functions are omitted for brevity.

```
function UPDATEPRETRACE($s_i$, $t$, $t^{\text{lastSpike}}$)
    $\Delta t \leftarrow t - t^{\text{lastSpike}}$ return $s_i \cdot \exp\left(-\frac{\Delta t}{tau}\right) + 1$

function APPLYPRESPIKE($w_{ij}$, $t$, $t_j$, $s_j$)
    $\Delta t \leftarrow t - t_j$
    if $\Delta t \neq 0$ then return applyDepression $\left(w_{ij}, s_j \cdot \exp\left(-\frac{\Delta t}{tau}\right)\right)$
    else return $w_{ij}$
```

Algorithm 2 Additive weight-dependence implementation. Equivalent *apply Potentiation* function is omitted for brevity.

```
function APPLYDEPRESSION($w_{ij}$, $d$) return $w_{ij} + A^{+} \cdot d$
```

Algorithm 3 The new SpiNNaker STDP algorithm

```
function PROCESSROW($t$, $flush$, $t^{\text{lastUpdate}}$, $t^{\text{lastSpike}}$, $s_i$, $synapses$)
    for all ($j$, $w_{ij}$, $d^A$, $d^D$) in $synapses$ do
        $history \leftarrow$ getHistoryEntries($j$, $t^{\text{lastUpdate}} + d^A - d^D$, $t + d^A - d^D$)

        for all ($t_j$, $s_j$) in history do
            $w_{ij} \leftarrow$ applyPostSpike($w_{ij}$, $t_j + d^D$, $t^{\text{lastSpike}} + d^A$, $s_i$)

        if not $flush$ then
            ($t_j$, $s_j$) $\leftarrow$ getLastHistoryEntry($t + d^A - d^D$)
            $w_{ij} \leftarrow$ applyPreSpike($w_{ij}$, $t + d^A$, $t_j + d^D$, $s_j$)
            addWeightToRingBuffer($w_{ij}$, $j$)

    if not $flush$ then
        $s_i \leftarrow$ updatePreTrace($s_i$, $t$, $t^{\text{lastSpike}}$)
        $t^{\text{lastSpike}} \leftarrow t$
    $t^{\text{lastUpdate}} \leftarrow t$
```

times (t_j) at which that neuron spiked between $t^{\text{lastUpdate}}$ and t and its state at that time (s_j) (taking into account the dendritic (D^D) and axonal (D^A) delays associated with each synapse). The algorithm continues by looping through each post-synaptic spike and calling the *applyPostSpike* function to apply the effect of the interaction between the post-synaptic spike and the pre-synaptic spike that occurred at $t^{\text{lastSpike}}$ to the synapse. If the update was instigated by a pre-synaptic spike rather than a flush, the *applyPreSpike* function is called to apply the effect of the interaction between the pre-synaptic spike and the most recent post-synaptic spike to the

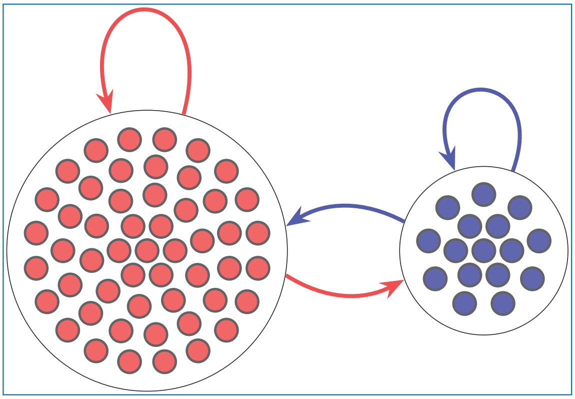


Figure 7.11. A random balanced network consisting of recurrently and reciprocally connected populations of excitatory neurons (red filled circles) and inhibitory neurons (blue filled circles). Excitatory connections are illustrated with red arrows and inhibitory connections with blue arrows.

synapse. Once all events are processed, the fully updated weight is added to the input ring buffer. If the update was instigated by a pre-synaptic spike rather than a flush, after all the synapses are processed, the pre-synaptic state stored in the header of the row (s_i) is updated by calling the *updatePreTrace* function and $t^{\text{lastSpike}}$ and $t^{\text{lastUpdate}}$ are set to the current time. If, however, the update was instigated by a flush event, only $t^{\text{lastUpdate}}$ is updated to the current time, meaning that the interactions between future post-synaptic events and the last pre-synaptic spike will continue to be calculated correctly.

7.2.4 Inhibitory Plasticity in Cortical Networks

One of the simplest models of a cortical network consists of a population of excitatory neurons and a smaller population of inhibitory neurons, sparsely connected with the recurrent and reciprocal synapses shown in Figure 7.11, that is, the random balanced network introduced in Chapter 4. Brunel [27] identified that these networks can operate in several well-defined regimes depending on the relative weights of the inhibitory and excitatory synapses. The asynchronous irregular regime has proved of particular interest as it matches firing rate statistics recorded in the neocortex [232] and responds rapidly to small changes in input making it an ideal substrate for computation [262]. However, it is unclear how the carefully balanced synaptic weights required to establish this regime are maintained in the brain. Vogels *et al.* [261] demonstrated that the asynchronous irregular regime can be

Algorithm 4 Inhibitory plasticity timing-dependence implementation. *updatePreTrace* and *updatePostTrace* functions are identical to those used by standard STDP and are therefore omitted for brevity.

```
function APPLYPRESPIKE($w_{ij}, t, t_j, s_j$)
    $\Delta t \leftarrow t - t_j$ return $applyPotentiation\left(w_{ij}, s_j \cdot \exp\left(-\frac{\Delta t}{tau}\right) - \alpha\right)$
function APPLYPOSTSPIKE($w_{ij}, t, t_i, s_i$)
    $\Delta t \leftarrow t - t_i$ return $applyPotentiation\left(w_{ij}, s_i \cdot \exp\left(-\frac{\Delta t}{tau}\right)\right)$
```

established using an STDP rule with the type of symmetrical kernel shown in Figure 7.5. We implemented this learning rule using the timing dependence functions defined in Algorithm 4 and used it to reproduce the results presented by Vogels *et al.* using a network of 2,000 excitatory and 500 inhibitory neurons with the parameters listed in Table 7.1.

Without inhibitory plasticity, the network remained in the synchronous regime shown in Figure 7.12(a) in which neurons spiked simultaneously at high rates. However, with inhibitory plasticity enabled on the connection between the inhibitory and the excitatory populations, the neural activity quickly stabilised and, as shown in Figure 7.12(b), the network entered an asynchronous irregular regime in which neurons spiked at a much lower rate.

7.2.5 The Effect of Weight Dependencies

In Section 7.2.1, we discussed how the choice of weight dependence affects the fit of STDP models to biological data. Rubin *et al.* [214] showed that different weight dependencies also result in different equilibrium distributions of synaptic weights when neurons with a biologically plausible number of synapses are stimulated with Poisson spike trains. Rubin *et al.* proved that a multiplicative weight dependence results in a unimodal distribution of weights, whereas an additive dependence results in a bimodal distribution.

To demonstrate the flexibility of the SpiNNaker STDP implementation presented here, we reproduced these results empirically using the simple PyNN model described in Table 7.2. The resultant weight distributions are plotted in Figure 7.13. Additive weight dependencies in PyNN specify hard upper and lower bounds and, as Rubin *et al.* predicted, the experiment using the additive weight dependence results in a weight distribution with modes centred at these bounds. Again, as Rubin *et al.* predicted, the experiment using the multiplicative weight dependence results in a unimodal weight distribution.

Table 7.1. Model description of the inhibitory plasticity network. After Nordlie [182]

Model summary

Populations	Excitatory, inhibitory
Connectivity	Probabilistic with 2% connection probability
Neuron model	LIF with exponential current inputs
Plasticity	Inhibitory plasticity (Vogels *et al.* [261])

Populations

Name	Elements	Size
Excitatory	LIF	2,000
Inhibitory	LIF	500

Connectivity

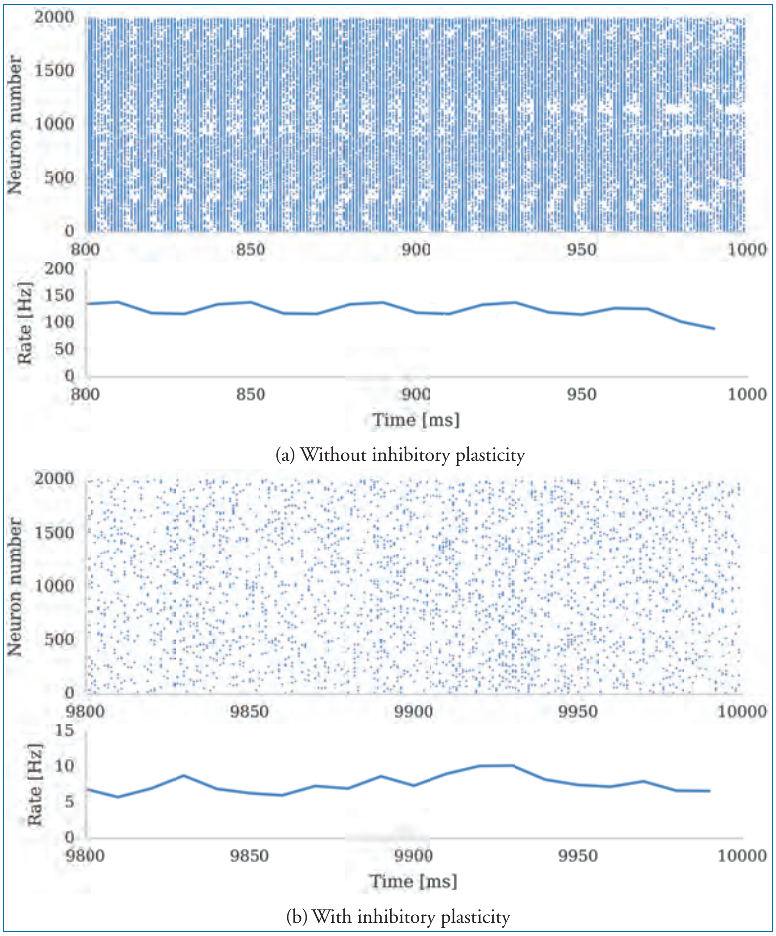
Source	Target	Weight
Excitatory	Inhibitory	0.03 nA
Excitatory	Excitatory	0.03 nA
Inhibitory	Inhibitory	0.3 nA
Inhibitory	Excitatory	0 nA

Neuron and synapse model

Type	LIF with exponential current inputs
Parameters	$g_L = 0.01$ µS leak conductance
	$C = 0.2$ nF membrane capacitance
	$V_{thresh} = -50$ mV threshold voltage
	$V_{reset} = V_{rest} = -60$ mV reset/resting voltage
	$\tau_{syn}^{exc} = 5$ ms excitatory synaptic time constant
	$\tau_{syn}^{inh} = 10$ ms inhibitory synaptic time constant

Plasticity

Type	Inhibitory plasticity (Vogels *et al.* [261])
	on inhibitory excitatory synapses
Parameters	$\rho = 0.12$ µS post-synaptic target firing rate
	$\tau = 20.0$ ms trace time constant
	η learning rate



(a) Without inhibitory plasticity

(b) With inhibitory plasticity

Figure 7.12. The effect of inhibitory plasticity on a random balanced network with 2,000 excitatory and 500 inhibitory neurons. Without inhibitory plasticity, the network is in a synchronous state with all neurons firing regularly at high rates. Inhibitory plasticity establishes the asynchronous irregular state with all neurons firing at approximately 10 Hz.

7.3 Voltage-Dependent Weight Update

Although highly successful, the STDP algorithm has some drawbacks. For example, if the simulator has no memory of pre- and post-synaptic spike times, the algorithm is difficult to implement; furthermore, if the post-synaptic neuron fails to spike, it could be that important information is lost. Bengio *et al.* [14] propose a plasticity rule which is compatible with STDP dynamics and could, in principle, be a way to link machine learning and neuroscience.

Table 7.2. Model description of the synaptic weight distribution network. After Nordlie [182].

Model summary		
Populations	Neurons, stimuli	
Connectivity	All-to-all	
Neuron model	LIF with exponential current inputs	
Plasticity	STDP	
Populations		
Name	Elements	Size
Neurons	LIF	1
Stimuli	Independent 15 Hz Poisson spike trains	1,000
Connectivity		
Source	Target	Weight
Stimuli	Neurons	Uniformly distributed between 0 nA to 0.01 nA
Neuron and synapse model		
Type	LIF with exponential current inputs	
Parameters	$g_L = 0.017\,\mu\text{S}$ leak conductance	
	$C = 0.17\,\text{nF}$ membrane capacitance	
	$V_{thresh} = -54\,\text{mV}$ threshold voltage	
	$V_{reset} = -60\,\text{mV}$ reset voltage	
	$V_{rest} = -74\,\text{mV}$ resting voltage	
	$\tau_{syn} = 5\,\text{ms}$ synaptic time constant	
Plasticity		
Type	STDP with additive or multiplicative weight dependence	
Parameters	$A^+ = 0.01$ potentiation rate	
	$A^- = 0.0105$ depression rate	
	$\tau_+ = 20.0\,\text{ms}$ trace time constant	
	$\tau_- = 20.0\,\text{ms}$ learning rate	

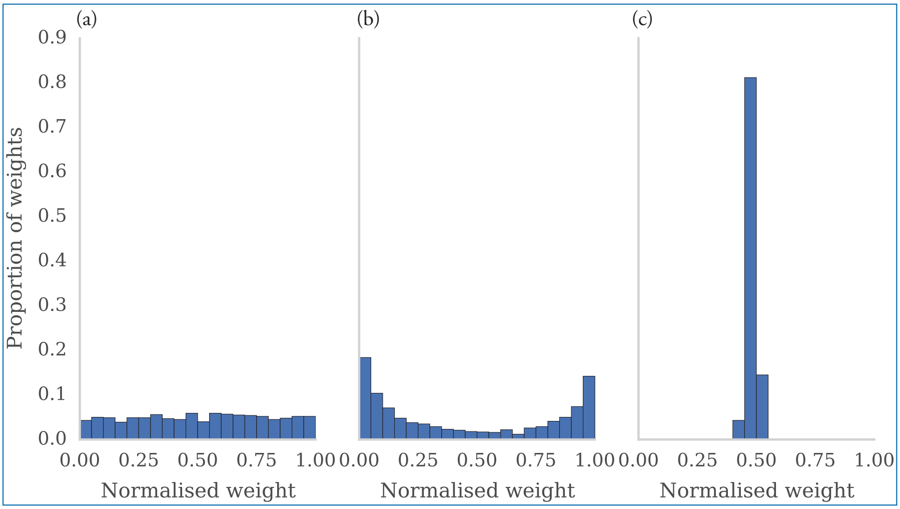


Figure 7.13. Histograms showing: (a) initial uniform distribution of synaptic weights and distribution of synaptic weights following; (b) STDP with additive weight dependence and (c) STDP with multiplicative weight dependence. Simulation consists of a single integrate-and-fire neuron with 1,000 independent 15 Hz Poisson spike sources providing synaptic input.

Table 7.3. Izhikevich neuron model parameters.

	a	**b**	**c**	**d**
Value	0.02	0.2	−65	8
Units		dimensionless		

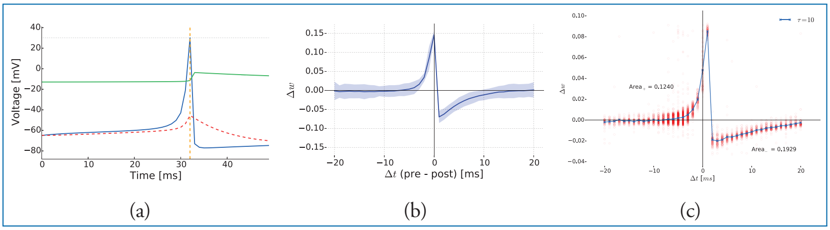
We implemented this rule using both LIF and Izhikevich neuron models [193]; we only show the results with the latter here. The parameters used in our experiments for the Izhikevich model are shown in Table 7.3. The behaviour of the neuron model when a continuous current is applied is illustrated in Figure 7.14(a).

The blue line depicts the neuron's membrane voltage (v), and the green line shows the behaviour of the auxiliary variable u. Since the membrane voltage is usually noisy, we filter it using the exponential smoothing technique [175]

$$\gamma = e^{-1/\tau_s}, \tag{7.14}$$

$$s(t) = (1-\gamma)v(t) + \gamma s(t-1); \tag{7.15}$$

where τ_s is the temporal constant for the filtering mechanism. The dashed red line is a low-passed version of the membrane voltage (s). The change in synaptic efficacy



(a) (b) (c)

Figure 7.14. Membrane voltage change as a proxy for weight updates. (a) Behaviour of an Izhikevich neuron to a step input; the blue line illustrates the membrane voltage, while the red dashed line shows a low-pass-filtered version of it. (b) shows how an average of weight changes behaves close to STDP when simulated in Python. (c) summarises the average of weight changes simulated on SpiNNaker.

is given by

$$\Delta w = \alpha \times \delta(t - t_{pre}) \times \frac{\Delta s(t)}{\Delta t} \tag{7.16}$$

where $\delta(\cdot)$ is the Kronecker delta function and α is the scaling factor and could be used as the learning rate. To test whether this adjustment to the original learning rule still produces similar results (i.e. STDP-compatible behaviour), we establish an experimental set-up similar to the one presented by Bengio *et al.* [14]. We simulate 5,000 neurons (using a home-brew implementation) which have a noisy current offset; random input spikes are generated at every time step, with a 20% and 5% probability, for excitatory and inhibitory types, respectively. All synapses are characterised as a 1 ms pulse response; weights for inhibitory synapses are fixed, while excitatory are plastic.

We then look for post-synaptic neuron spikes and collect weight change statistics in a ±20 ms temporal window in 1 ms time steps. We compute the average change for each time step; Figure 7.14(b) depicts the resulting averages for voltage changes.

We performed a similar experiment using the SpiNNaker implementation and computed the average of generated data points which gives rise to an STDP-like curve (Figure 7.14(c)). A major difference is that the curve gets shifted 1 ms to the right; this is because weight changes are computed as soon as a spike arrives at the post-synaptic core but applied a time step later.

7.3.1 Results

To test the viability of using this learning rule to capture the statistics coming from visual patterns, we set experiments with networks based on a SWTA circuit.

Unsupervised: We first explore an unsupervised learning procedure. The network for this experiment is presented in Figure 7.15 and is composed of an input layer

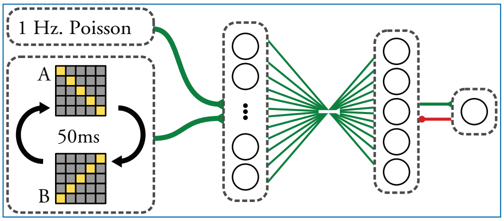


Figure 7.15. SWTA network with visual pattern as input. A 5 × 5 pixel/neuron array is given an input which corresponds to the two main diagonals alternated with a 50 ms delay between them; it is also provided with a 1 Hz noise with a Poisson distribution. This array is connected with plastic connections to 5 target neurons, which in turn are in a SWTA circuit.

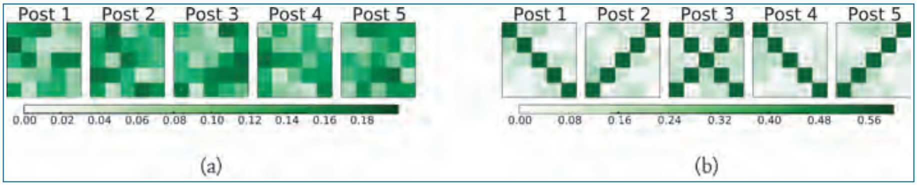


Figure 7.16. Weight changes after alternating pattern simulation. (a) shows the input weights for each target neuron, and these were set at random with a uniform distribution [0.05, 0.2). (b) By the end of the simulation, some neurons have specialised for a pattern as shown by the weights.

and an output layer. The input layer consists of 25 input neurons which serve as a relay for noise and input patterns. The network is given alternating visual patterns (diagonals) as an input; a 1 Hz Poisson noise source is added to the input in order to favour diversity of learning in the output layer. The output layer consists of five neurons which feed a single inhibitory neuron, the latter will reduce the chances of spiking for neurons in the output layer.

The synaptic weights from the input to the output layer were plastic and randomly initialised as shown in Figure 7.16(a). After multiple exposures to the input patterns, the synapses corresponding to the inputs get potentiated (Figure 7.16(b)). Particularly, neurons 1 and 4 become specialised on one pattern while 2 and 5 to the other.

Supervised: To establish a supervised learning regime, we take inspiration from biology and add a special signal which will enhance Long-Term Potentiation (LTP) when present. Biological neurons get further depolarised when N-Methyl-D-aspartic Acid (NMDA) is received and the membrane potential is above a certain level [160]. Implementing a similar behaviour (slow decay and voltage dependence) on SpiNNaker required altering how input current is computed by default,

$$I = \sum I_{+} - \sum I_{-} \tag{7.17}$$

where I is the total input current given to the neuron which consists of I_{+}, the excitatory inputs, and I_{-}, the inhibitory ones. With this scheme, it is not possible to condition the acceptance of current provided by NMDA activation influx, I_{ϕ}, and thus, Equation 7.17 will be modified to

$$I = \sum I_{+} - \sum I_{-} + \sum I_{\phi} \tag{7.18}$$

We control the activation of NMDA receptor channels in two ways. The first is through a special ϕ spike which emulates a gating mechanism of the channel (i.e. the presence of both glutamate and glycine [66, 179]). We also use this event to model the current (I_{ϕ}) created by additional positive ions passing through the opened channel. Secondly, I_{ϕ} is allowed to pass into the neuron only when the membrane voltage is above the threshold V_{ϕ}:

$$I_{\phi} = \begin{cases} I_{\phi} & \text{if } V_m > V_{\phi} \text{ or } t - t_{\phi} < T_{\phi} \\ 0 & \text{otherwise} \end{cases} \tag{7.19}$$

Furthermore, to mimic the time it takes to close the channels, we added an inertia-like mechanism ($t - t_{\phi} < T_{\phi}$ in Equation 7.19) which keeps I_{ϕ} current flowing for at least T_{ϕ} simulation steps. To achieve this, we subtract the time at which a ϕ spike arrived (t_{ϕ}) from the current simulation step and, if the temporal difference is smaller than the inertia window T_{ϕ}, the current is allowed to keep flowing. To test the supervision mechanism we used a similar network setup to that in the unsupervised case though we need two 'instances' of the network (Figure 7.17); one will 'supervise' the other. Each instance has five output neurons which are each assigned to an input pattern. During the experiment, neurons 1 and 2 of the bottom output population (Figure 7.17) were assigned to learn pattern 1 (forward diagonal), and neurons 3 and 4 were set to learn pattern 2 (back diagonal). The way we induce neurons to learn a particular pattern is to send a ϕ spike 5 ms before the pattern is shown to the corresponding neuron. Neurons in the bottom population connect to the neurons in the top population in a one-to-one manner through the lateral ϕ channel. For this experiment, we used Izhikevich neurons for the output and inhibitory populations which were configured according to the parameters shown in Table 7.4.

Figure 7.18 shows synaptic efficacies at the beginning and end of the training. Each square depicts the values of incoming synapses to a post-synaptic neuron, the top row of squares corresponds to the student population and the bottom row

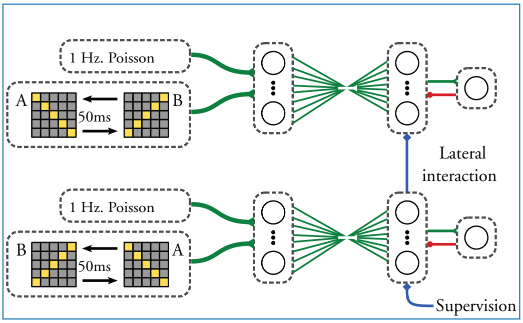


Figure 7.17. Supervision architecture.

Table 7.4. Neuron parameters for pattern learning using ϕ.

	a	b	c	d	τ_u	τ_{exc}	τ_{inh}	τ_ϕ	θ_ϕ
Excitatory	0.01	0.2	−65	8	10	2	2	50	−75
Inhibitory	0.2	0.26	−65	0	–	1	1	–	–
Units	dimensionless				ms	ms	ms	ms	mV

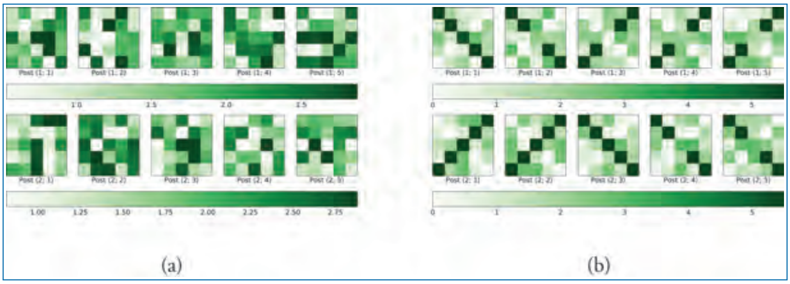


Figure 7.18. Weights at start and end of training using an NMDA-like signal ϕ. (a) Synaptic efficacies are set to random values initially. (b) After ~30 min of simulation, the weights favour the assigned input patterns.

corresponds to the teacher population. At the start (Figure 7.18(a)), weights are assigned randomly.

During training, a supervision ϕ spike is given to neurons ⟨2, 1⟩, ⟨2, 2⟩, ⟨2, 4⟩ and ⟨2, 5⟩ right before one of the patterns reaches them. Neurons ⟨2, 1⟩ and ⟨2, 2⟩ are assigned the forward diagonal pattern, while neurons ⟨2, 4⟩ and ⟨2, 5⟩ are assigned the backward diagonal pattern. Neuron (2, 3) is allowed to learn its input without any supervision. Since patterns for the student population are delayed (and inverted), the ϕ spikes coming from the teacher enforce neurons in the student population to learn a particular input (backward diagonal for ⟨1; 1⟩ and ⟨1, 2⟩, forward diagonal for $\langle 1, 4\rangle$ and $\langle 1, 5\rangle$). Figure 7.18(b) shows synaptic efficacies at the end of training, note that the largest weights correspond to the assigned patterns. While the task to learn here is a simple one, the behaviour of the network could be seen as self-supervision and could be applied to networks whose neurons learn parts of a larger problem.

7.4 Neuromodulated STDP

Traditionally, simple models of SNNs have used two types of synapse: excitatory (positive) and inhibitory (negative). These drive changes to the membrane voltage and, indirectly, can produce weight changes [74].

In biological neural networks, there are, in addition, neurotransmitters and neuromodulators that may alter learning processes. Research on dopamine interaction shows that it could be crucial for reinforcement learning as it has been identified as a control signal for large regions of the brain.

Furthermore, there are additional cells involved in synapse function – *astrocytes* – which are usually characterised as 'maintainers' in the central nervous system as they keep ionic concentrations stable, form scar tissue on damaged regions and aid energy transfer [231]. Scientists are putting more effort to understand the role of astrocytes as learning modulators [78]. Research shows that these cells are also involved in the regulation of current, frequency, short- and long-term plasticity, and synapse formation and removal [91, 189, 253, 254].

Having a third component modifies Hebbian-based weight updates, and the rule will now depend on the state of three factors (Figure 7.19):

$$\Delta w_{pre,post}(t) \propto h(s(pre), s(post), s(third)) \tag{7.20}$$

where $s(\cdot)$ indicates the activity or state. If only the spike time is considered as the state (as is the case for STDP),

$$\Delta w_{pre,post}(t) \propto h\left(t, t_{pre}, t_{post}, t_{third}\right) \tag{7.21}$$

where t_x is the time at which a spike from neuron x was perceived by the post-synaptic neuron.

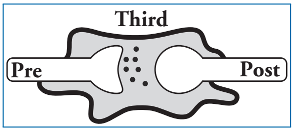


Figure 7.19. Cartoon of third factor interaction on plasticity.

Ponulak and Kasinski [200] introduced an STDP-like rule with a third factor (ReSuMe), in which the extra input is used to get the post-synaptic neuron to spike at a particular time.

When $t_{third} - t_{pre} > 0$, a weight increment (Δ_+) is applied to synapses, whereas a weight depression (Δ_-) is applied when $t_{post} - t_{pre} > 0$. If the post-synaptic neuron activates at the desired time $\left(t_{third} = t_{post}\right)$, depression will be equal to potentiation and the total weight change will be zero.

Gardner and Grüning [70] developed a three-factor learning rule whose purpose is, also, to learn spike times; to do this, the third input to the synapse carries a 'temporal target' signal and will alter the magnitude and direction of the weight change. The main difference from the ReSuMe rule is that this requires, additionally, a low-pass filtered version of the error. The temporal error is to modify the effect a single pre-synaptic spike has on post-synaptic neuron activity. The filtered version of the error can be seen as an 'accumulation' activity for a time window ($\approx$ 10 ms).

While the previously mentioned rules make use of a third factor, they remain biologically implausible as a synapse is unlikely to be able to keep track of exact times. In this context, we can see the neurotransmitter dopamine as a global error signal or a modulator that enables learning after the previous activity in the network led to a reward-worthy action [248].

Other modulators (e.g. serotonin and noradrenaline) could guide plasticity through attention-like mechanisms. These are thought to be local signals – as opposed to dopamine – and may represent feedback and/or lateral interaction [211].

Models of modulated synaptic plasticity have been developed and in general follow

$$\Delta w_{pre,post}(t) \propto g(t, t_{third}) \times f(t, t_{pre}, t_{post}) \tag{7.22}$$

where f is the regular plasticity function (e.g. STDP) and g is the, usually, decaying response of the modulatory input.

7.4.1 Eligibility Traces/Synapse Tagging

For reinforcement learning, a history of the plasticity function is required, usually called an *eligibility trace*. The intuition behind this mechanism is that events in the world occur at a lower speed than spike interactions and behaviour can be rewarded (or punished) only after it has happened. Furthermore, researchers have found some evidence of eligibility traces in biology [59, 75].

In mathematical models, eligibility traces 'store' weight updates for a long period, until a signal arrives at the synapse which triggers 'application' of the current state

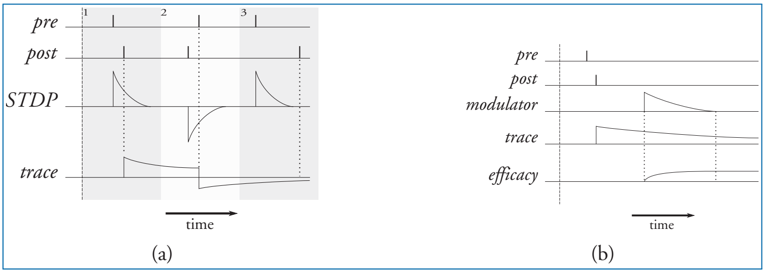


Figure 7.20. Eligibility trace and modulated synaptic efficacy changes.

of the trace [60, 118]. The signal could be dopamine, or another neuromodulator with slower dynamics, decaying on the scale of hundreds of milliseconds.

In Figure 7.20(a), the trace labelled STDP shows the weight change function given the inputs shown in rows *pre* and *post*. Eligibility traces are formed by $\langle pre,\ post\rangle$ spike pairs which are illustrated in zones 1 and 2 in Figure 7.20(a); these cause accumulation of weight changes driven by STDP curves. Since STDP interactions depend on the time at which the pre- and post-synaptic neurons spiked, if these times are sufficiently far apart in time, no weight change is added to the eligibility trace (compare zone 3 with zone 2 in Figure 7.20(a)).

Eligibility traces have much slower dynamics than STDP interactions as illustrated in Figure 7.20(a); the curve in row *trace* decays much slower than any of the curves in row *STDP*. The low decay rate is useful to keep track of how temporally distant weight changes contributed to a particular behaviour.

The modulating neurotransmitter (*modulator* curve in Figure 7.20(b)) also has slower dynamics than STDP, but not as slow as eligibility traces. Weight changes are only applied when the third signal is present; this is modelled as a multiplicative effect

$$\Delta weight(t) \propto modulator(t) \times trace(t), \text{ or,} \tag{7.23}$$

$$\frac{dw(t)}{dt} = m(t) \times c(t). \tag{7.24}$$

We implemented the dopamine-based modulated plasticity model as proposed by Izhikevich on the SpiNNaker machine [165]. Eligibility trace dynamics are described by the following equation:

$$\frac{dc(t)}{dt} = -\frac{c(t)}{\tau_c} + STDP(\tau_{-/+})\delta(t - t_{pre/post}); \tag{7.25}$$

where $c(t)$ is the state of the eligibility trace; $STDP(\tau_{-/+})$ is the value from STDP (Figure 7.20(a)) curves and $\delta(t - t_{pre/post})$ is the Dirac delta function. Similarly, the modulator is governed by

$$\frac{dm(t)}{dt} = -\frac{m(t)}{\tau_m} + M(t)\delta(t - t_{mod}) \tag{7.26}$$

where $m(t)$ is the current state of the 'local' modulating transmitter and $M(t)$ is the concentration of the 'external' modulatory signal. In both cases, incoming signals create an instantaneous change (spike), thus the use of Dirac delta functions.

Since SpiNNaker is an event-driven computation platform, these equations required modifications. Weight changes are performed when a spike arrives at a post-synaptic core, and thus, the implemented weight update rule is:

$$\Delta w(t) = \frac{1}{-\frac{1}{\tau_c} - \frac{1}{\tau_m}} c(t_{lc}) m(t_{lm}) \left[e^{-\left(\frac{t-t_{lc}}{\tau_c}\right) - \left(\frac{t-t_{lm}}{\tau_m}\right)} \right]_{t_{lw}}^{t}, \tag{7.27}$$

where lc, lm and lw subscripts indicate the time (event) at which the last eligibility trace, modulator and weight updates were performed, respectively. As two different spike 'types' can be received, weight updates will be performed either at $t_{lw} = t_{lc}$ or $t_{lw} = t_{lm}$. The evaluation of the squared brackets in Equation 7.27 is done as in definite integrals.

7.4.2 Credit Assignment

We tested the learning rule replicating an experiment which was originally designed by Izhikevich [117]. The goal is to identify a group of neurons spiking amongst noise and reinforce the groups' connections while keeping other neurons' connectivities in a lower weight range.

The neural network for this experiment consists of 1,000 neurons, divided into two populations. They emit either excitatory or inhibitory signals (*Exc* and *Inh*, respectively). Each neuron connects to others at random with a 10% probability, regardless of the neuron population 'type' (red and green lines in Figure 7.21).

Within the excitatory population, we create groups of 50 neurons (5% of the total) chosen at random; the first group (S_1) is chosen as the pattern to search. We stimulate each group at random, maintaining a maximum of 5 groups spiking per second. Additionally, we inject 'background' Poisson noise at 10 Hz, and this puts neurons in a biologically plausible setting. In terms of the credit assignment problem, we try to identify a signal (S_1 group activity) in a noisy environment, generated by other groups and random activity incited by background noise.

To search for group S_1's particular activity, we connect a modulator input (dopamine-like) to the excitatory-to-excitatory connections. Whenever group S_1 is stimulated, we send a dopamine pulse with a random delay in the range [0, 1) seconds; this will act as the teaching signal.

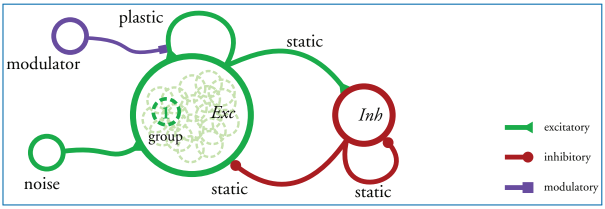


Figure 7.21. Credit assignment experiment network – yet another random balanced network.

Table 7.5. LIF neuron model parameters for the credit assignment experiment.

	C_m	I_{offset}	τ_m	τ_{refrac}	τ_{syn_E}	τ_{syn_I}	V_{reset}	V_{rest}	V_{thresh}
Value (Exc)	0.3	0.0	10	4	1	1	−70	−65	−55.4
Value (Inh)	0.3	0.005	10	2	1	1	−70	−65	−56.4
Units	nF	nA	ms	ms	ms	ms	mV	mV	mV

Table 7.6. STDP parameters for the credit assignment experiment.

	A_+	A_-	τ_+	τ_-	τ_c	τ_d
Value	1	1	10	12	1,000	200
Units	–	–	ms	ms	ms	ms

For this experiment, we use standard LIF neurons whose parameters promote higher activity from the elements in the inhibitory group. High excitability is achieved by adding a small base current, reducing the distance between the resting voltage and the threshold value, and reducing the refractory period (see Table 7.5). We used exponentially decaying, current-based synapses whose temporal constants (τ_{syn_X}) are set to 1 ms to further approximate the original experiment.

The learning algorithm was parametrised so that the area for LTP is 20% greater than the area for Long-Term Depression (LTD) (A_+, A_-, τ_+, τ_- in Table 7.6). Dopamine interaction was characterised in a biologically plausible manner: the temporal constant for the eligibility trace, τ_c, is 1,000 ms, which implies the network will learn events that occurred up to a second ago; the temporal constant for dopamine, τ_d, is 200 ms according to biological evidence.

We ran the experiment for about 1.5 hours of biological real time. At approximately 70 minutes, the weights from group S_1 to other excitatory neurons are sufficiently large to be visibly noticeable in a raster plot.

The evolution of the average weight in group S_1 is shown in Figure 7.22(a) as a green line, which presents an exponential growth. We can also observe the average

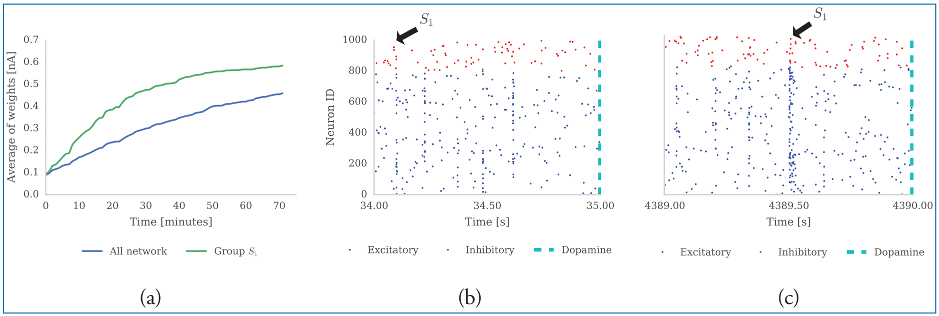


Figure 7.22. Credit assignment experiment network behaviour. Discussed in detail by Mikaitis *et al.* [165].

weight value for every connection in the network as the experiment progresses (blue line), it grows but more slowly and it is expected to stabilize. Spiking behaviour for all groups is similar at the start of the experiment; this can be seen as correlated vertical dots in Figure 7.22(b).

By the end of the experiment, connections which originate from group S_1 are at such a value that most post-synaptic neurons will spike when a neuron in the group is active. This is shown in the middle of Figure 7.22(c) as a burst of activity. Although the network still responds to other patterns, it is now tuned to emit a higher response to the S_1 pattern. This has been observed in cortical regions; for example, a column in V1 will show a response to many oriented bars as inputs, but it presents the maximum spike rate for a particular orientation.

7.5 Structural Plasticity

Mammalian brains have evolved in an ever-changing environment and, as such, are equipped with a wealth of learning mechanisms. One such mechanism that has been implemented on SpiNNaker is usually referred to as structural plasticity [21]. This mechanism relies on the structural changes in the network guided by some measure. Since SpiNNaker focuses on simulating neurons without morphological detail, structural plasticity is equated with changes in connectivity between neurons – synaptic rewiring.

SpiNNaker is particularly well suited for supporting structural plasticity as a learning mechanism. This software-driven neuromorphic platform is sufficiently flexible to support rewiring in parallel with STDP while still maintaining the real-time wall clock [23]. Furthermore, synaptic rewiring can be used to ensure optimal resource use in a system that is comparatively memory and compute cycle starved [73]. Intuitively, if the rewiring process can be made to maintain a certain synaptic capacity usage while preferentially discarding 'useless' synapses, then the system can search for the optimal synaptic configuration that fits within the constraints of the system.

SpiNNaker's real-time constraint also means that synaptic rewiring simulations occurring at a slower time scale compared to other neural and synaptic processes can be monitored over longer time scales.

The model chosen for translation onto SpiNNaker was developed by Bamford *et al.* [8]. Their model had some desirable features and posed some interesting challenges. Feature wise, the rewiring rules allowed for synaptic formation and removal driven by Euclidean distance and synaptic weight, respectively.

A certain number of rewiring attempts are performed every simulated second. Each neuron in the network has a limited number of potential synaptic contact points. Attempts either follow the formation or removal rules dependent on the existence of a synapse at a considered contact point.

Formations favour neurons that are relatively close in space, which also represents the first challenge for SpiNNaker. This was the first time that SpiNNaker neural models required spatial information for the simulated neurons. A new, full-strength, connection is formed with a partner neuron that has fired recently if

$$r < p_{form} e^{-\frac{\delta^2}{2\sigma_{form}^2}} \tag{7.28}$$

where r is a random number sampled from a uniform distribution in the interval $[0, 1)$, p_{form} is the peak formation probability, δ is the distance between the two cells and σ_{form}^2 is the variance of the receptive field. The result is a Gaussian distribution of formed synapses around the ideal target site, that is, around the target neuron where $\delta = 0$.

If the randomly selected synaptic contact is in use (a synapse already exists), the removal rule is followed. For implementation efficiency and because of the nature of the synaptic plasticity rule (weight-independent STDP), a weight threshold θ_g is selected as half of the maximum allowed weight ($\theta_g = \frac{1}{2} g_{max}$). A synapse is removed if

$$r < p_{elim} \quad \text{where} \quad p_{elim} = \begin{cases} p_{elim-dep} & \text{for } g_{syn} < \theta_g \\ p_{elim-pot} & \text{for } g_{syn} \geq \theta_g \end{cases} \tag{7.29}$$

where r is a random number sampled from a uniform distribution in the interval $[0, 1)$, $p_{elim-dep}$ is the elimination probability used when a synapse is depressed, $p_{elim-pot}$ is the elimination probability used when a synapse is potentiated and g_{syn} is the weight of the synapse under consideration for removal.

The first application of this model of structural plasticity (not to be confused with The Model for Structural Plasticity – MSP – proposed by Butz and van Ooyen

[29]) was to replicate and expand on previous results regarding topographic map formation (Section 7.5.1)

Section 7.5.3 shows that choosing a partner for formation from the set of recently active cells is a powerful mechanism. In the context of MNIST handwritten digit classification, this feature allows for decent accuracy and recall scores in the absence of weight changes as long as the input is encoded using spiking rates. Further, Section 7.5.4 reveals the importance of visualisation in identifying the cause of aberrant behaviour using this particular feature as its main example.

A final application of these synaptic rewiring rules in described in Section 7.5.5 where it is shown to perform elementary motion decomposition after the application of additional minor enhancements.

An alternative formulation based upon information theory, sparse codes and their congruence with recent results about the behaviour of clustered synapses in real dendritic trees is described briefly by Hopkins *et al.* [103].

7.5.1 Topographic Map Formation

A widely observed principle in biological brains is the use of topographic maps, wherein two-dimensional topological (though not necessarily scale) relationships are preserved in projections from one brain region to another.

Neural topographic maps consist of layers of neurons whose reaction to afferent (incoming) stimuli changes with area (Figure 7.23). Such an organisation is characterised by the preservation of neighbour activity from the source to the target layer and provides several advantages in terms of wiring and information processing and integration. Wiring is optimised since neurons generally have limited receptive fields and tend to be interested in spatially clustered locations. As an example, orientation-selective neurons, such as those present in primary visual cortex, are required to have afferents from small regions of the total visual receptive field, thus a topographic organisation ensures that neurons only connect to their immediate neighbours and have limited interaction with those which are further away. More importantly, when neurons form multiple aligned maps, each receiving information from a different modality, they exhibit multisensory facilitation; their response is supra-linear if they receive synchronous stimuli from the same area of space arriving from different modalities. This is the case in the Superior Colliculus, a brain structure that integrates signals from multiple senses and also guides adaptive motor responses [126].

Topographic projections are widespread in the mammalian cortex [123]. Their development has been explored through simulation, with and without spiking neurons, and involving both synaptic plasticity [131, 233, 264] and synaptic rewiring [8]. The latter example has been modelled on SpiNNaker. It is with

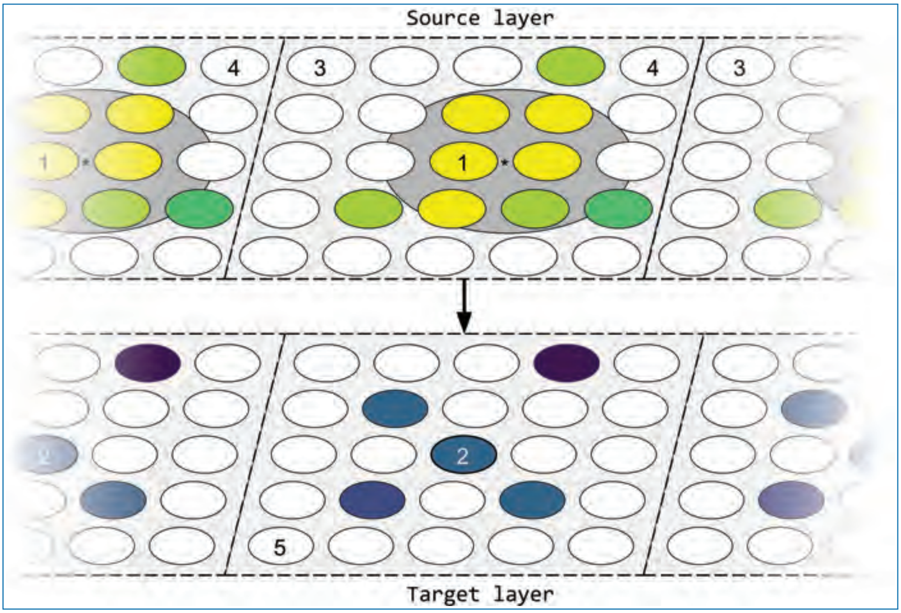


Figure 7.23. Topographic maps. Neuron (2) in the target layer has a receptive field formed by connections from the source layer (feed-forward) as well as connections from within the target layer (lateral). These connections are centred around the spatially closest neuron, that is neuron (1) in the case of feed-forward connections. Connections from more distant neurons are likely to be weaker (indicated by a darker colour).

that model we suggest an architecture capable of handwritten digit classification through supervised learning enabled by the construction of the training architecture.

In short, the suggested model involves the co-operation of two types of mechanisms: activity-independent and activity-dependent mechanisms. The former is represented by the formation rule: it relies on the distance between potential partnering neurons to create a new synapse – spatially clustered neurons will tend to form more connections than neurons that are spatially distant. The latter is composed of two mechanisms: spike-timing-dependent plasticity (STDP) and a removal rule for the synaptic rewiring mechanism. STDP, utilising local spiking information, modifies the weights of synapses connecting neurons together, while the elimination rule preferentially removes those synapses that are depressed – synapses that carry 'useful' patterns or sub-sets of patterns to neurons will tend to be re-inforced, thus are more stable in the long term. Conversely, synapses that usually transmit what amounts to noise will be silenced and are more likely to be pruned. All of the aforementioned mechanisms operate continuously, at a fixed rate, on a population of neurons.

Topographic map quality is assessed using the measures defined by Bamford *et al.* [8]: the spread of the mean receptive field σ_{aff} (grey circular area presented in Figure 7.23) and the Absolute Deviation (AD) of the mean receptive field from its ideal location (the centre of the circular area presented in Figure 7.23 as a star). The former relies on the search for the location around which afferent synapses have the lowest weighted variance (σ_{aff}^2). This is a move away from the centre of mass measurement used by Elliott and Shadbolt [55] for identifying the preferred location of a receptive field. As a result, the centre of the receptive field that is being examined is the location that minimises the weighted standard deviation, computed as follows:

$$\sigma_{aff} = \sqrt{\frac{\sum\limits_i w_i |\vec{p}_{xi}|^2}{\sum\limits_i w_i}} \tag{7.30}$$

where i loops over synapses, x is a candidate preferred location, $|\vec{p}_{xi}|$ is the minimum distance from the candidate location to the afferent for synapse i and w_i is the weight of the synapse. The candidate preferred location x has been implemented with an iterative search over each whole number location in each direction followed by a further iteration, this time in increments of 0.1 units. Thus, the preferred location $\vec{x}$ of a receptive field is given by the function: $\underset{\vec{x}}{\operatorname{argmin}}\, \sigma_{aff}$.

Once the preferred location of each neuron is computed, taking the mean distance from the ideal location of each preferred location results in a mean AD for the projection. We report both mean AD and mean σ_{aff} computed with and without taking into account connections weights. $\sigma_{aff-weight}$ and AD_{weight} are computed using synaptic weights g_{syn}, while $\sigma_{aff-conn}$ and AD_{conn} are designed to consider the effect of rewiring on the connectivity, and thus, synaptic weights are considered unitary.

These metrics are considered in three types of experiments:

- Case 1: STDP and rewiring operating simultaneously in the presence of correlated input
- Case 2: STDP but no rewiring in the presence of correlated input
- Case 3: STDP and rewiring without correlated input

Modelling developmental formation of topographic maps using the presented mechanisms could yield some interesting results with regard to total speed of simulation and could be used to validate whether the network can still generate good-quality topographic maps. Regarding the former, simulations with far more neurons and synapses could benefit from not requiring a fixed connectivity be

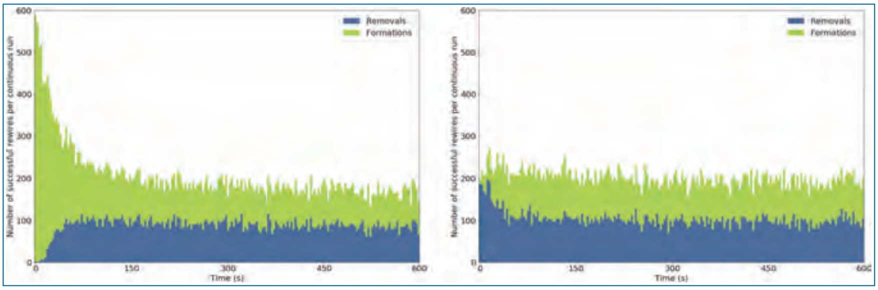


Figure 7.24. Stacked bar chart of formations and removals over time within one simulation. Left: evolution of the network starting from no connections; right: evolution of the network starting from a sensible initial connectivity. The number of formations or removals is aggregated into 3-second chunks.

loaded onto SpiNNaker – the process of interacting with SpiNNaker for loading or unloading data is currently the main bottleneck. The latter is meant as a validation for the model, but also to gain additional insights into its operation, specifically whether maps formed through a process of simulated development react differently to maps that are considered in their 'adult', fully-formed state in need of refinement.

Results from this simulation are presented in tabular form (Table 7.8), with the addition of longitudinal snapshots into the behaviour of the network and mean receptive field spread and drift, as well as a comparison between different initial connectivity types (topographic, random percentage-based and minimal). In the initial stages of the simulation, the σ_{aff} and AD are almost zero due to the lack of connections, but they steadily increase with the massive addition of new synapses. Figure 7.24 shows a side-by-side comparison of the number of rewires between early development and adult refinement. The developmental model initially sees a large number of synapses being formed until an equilibrium is reached at around 10% connectivity. A 10% connectivity is also achieved when starting the network from an adult configuration. This does not mean that every set of parameters will yield the same result. In this case, and all the others in this work, we locked the maximum fan-in for target layer neurons to 32, or 12% connectivity. As a result, the network is bound to have at most that connectivity and at least half that, or 6%, if formation and removal occur with equal probability.

Table 7.8 shows the final, single-trial, results for a network identical to previous experiments, but with a run time of 600 seconds. This ensures the networks have a chance to converge on a value of σ_{aff} and AD. A comparison between Tables 7.7 and 7.8 shows similar results for case 1, but significantly better results for Case 3. These differences are summarised in Figure 7.25. σ_{aff} and AD were computed at the end of three simulations differing only in the initial connectivity: an initial rough topographic mapping as in the previous experiments, a random

Table 7.7. Simulation results presented in a similar fashion to Bamford *et al.* [8] (Table 2) for three cases, all of which incorporate synaptic plasticity. Case 1 consists of a network in which both synaptic rewiring and input correlations are present. Case 2 does not integrate synaptic rewiring, but still has input correlations, while case 3 relies solely on synaptic rewiring to generate sensible topographic maps.

Case	**1**	**2**	**3**
Target neuron mean spike rate	21.15 Hz	20.11 Hz	9.31 Hz
Final mean feedforward fan-in per target neuron	15.91	N/A	11.87
Weight proportion of maximum	0.83	0.72	0.62
Mean $\sigma_{aff-init}$	2.35	2.35	2.35
Mean $\sigma_{aff-fin-conn-shuf}$	2.33	N/A	2.31
Mean $\sigma_{aff-fin-conn}$	1.62	2.35	1.85
p(WSR $\sigma_{aff-fin-conn}$ vs. $\sigma_{aff-fin-conn-shuf}$)	2.80×10^{-43}	N/A	3.65×10^{-27}
Mean $\sigma_{aff-fin-weight-shuf}$	1.61	2.32	1.78
Mean $\sigma_{aff-fin-weight}$	1.49	1.92	1.57
p(WSR $\sigma_{aff-fin-weight}$ vs. $\sigma_{aff-fin-weight-shuf}$)	4.03×10^{-33}	4.02×10^{-43}	1.44×10^{-21}
Mean AD_{init}	0.81	0.81	0.81
Mean $AD_{fin-conn-shuf}$	0.82	N/A	1.09
Mean $AD_{fin-conn}$	0.77	0.81	0.91
p(WSR $AD_{fin-conn}$ vs. $AD_{fin-conn-shuf}$)	0.39	N/A	0.002
Mean $AD_{fin-weight-shuf}$	0.79	0.92	1.04
Mean $AD_{fin-weight}$	0.85	0.79	1.07
p(WSR $AD_{fin-weight}$ vs. $AD_{fin-weight-shuf}$)	0.0002	0.0001	0.58

10% initial connectivity balanced between feedforward and lateral, and almost no initial connectivity (in practice, one-to-one connectivity was used due to software limitations). We do not simulate the case without synaptic rewiring, as the results would be severely impacted by the lack of rough initial topographic mapping. The

Table 7.8. Results for modelling topographic map formation from development (minimal initial connectivity).

Case	**1**	**3**
Target neuron mean spike rate	18.49 Hz	9.80 Hz
Final mean fan in / target neuron	17.69	11.92
Weight proportion of maximum	0.83	0.63
Mean $\sigma_{aff-init}$	0	0
Mean $\sigma_{aff-fin-conn-shuf}$	2.39	2.30
Mean $\sigma_{aff-fin-conn}$	1.56	1.67
p(WSR $\sigma_{aff-fin-conn}$ vs. $\sigma_{aff-fin-conn-shuf}$)	1.14×10^{-43}	2.27×10^{-35}
Mean $\sigma_{aff-fin-weight-shuf}$	1.56	1.59
Mean $\sigma_{aff-fin-weight}$	1.44	1.26
p(WSR $\sigma_{aff-fin-weight}$ vs. $\sigma_{aff-fin-weight-shuf}$)	1.25×10^{-36}	6.21×10^{-31}
Mean AD_{init}	0	0
Mean $AD_{fin-conn-shuf}$	0.79	0.99
Mean $AD_{fin-conn}$	0.64	0.85
p(WSR $AD_{fin-conn}$ vs. $AD_{fin-conn-shuf}$)	1.3×10^{-4}	0.01
Mean $AD_{fin-weight-shuf}$	0.66	1.02
Mean $AD_{fin-weight}$	0.65	0.96
p(WSR $AD_{fin-weight}$ vs. $AD_{fin-weight-shuf}$)	0.84	0.51

final mean value of AD is improved in both experiments involving initial non-topographic connectivity and σ_{aff} is improved when the network starts with no initial connectivity.

7.5.2 Stable Mappings Arise from Lateral Inhibition

Following the previous results, coupled with the tendencies exhibited by the results generated for Case 3 in Figure 7.26, further simulations have been run to confirm whether input correlations are necessary to generate stable topographic maps. Our results are inconclusive as although the weighted AD of the mean receptive field increased to 2.25, or 1.64 if only considering connectivity, the final

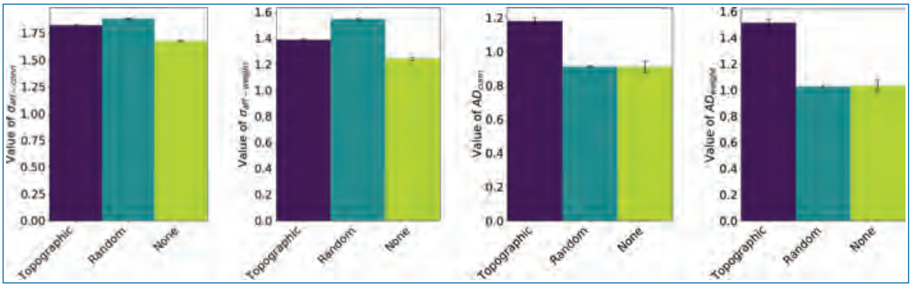


Figure 7.25. Comparison of final values for σ_{aff} and AD in the case where input correlations are absent (Case 3). Three types of networks have each been run 10 times (to generate the standard error of the mean), each starting with a different initial connectivity: an initial rough topographic mapping as in the previous experiments, a random 10% connectivity (5% feedforward, 5% lateral) and almost no connectivity (one-to-one connectivity used due to software limitations).

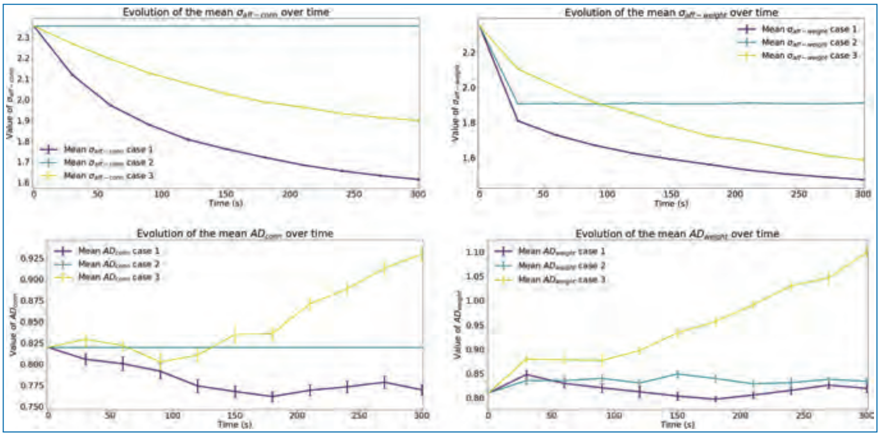


Figure 7.26. Evolution of results of interest. The top row shows the evolution of the mean spread of receptive fields over time, considering both unitary weights ($\sigma_{aff-conn}$) and actual weights ($\sigma_{aff-weight}$) at that point in time. The bottom row shows the evolution of the mean absolute deviation of the receptive fields considering connectivity (AD_{conn}) and weighted connectivity (AD_{weight}). Error bars represent the standard error of the mean.

mean number of feedforward synapses reduces to around 5, making these statistics unusable.

We hypothesise that since the feedforward is reduced so heavily, both in terms of connectivity (on average 5 incoming connections for each target-layer neuron) and in terms of synaptic weights (0.3 of maximum possible for the present connectivity) that the lateral connections within the target layer drive the comparatively high activity in the target layer (Figure 7.27). This, in turn, causes the pre-synaptic

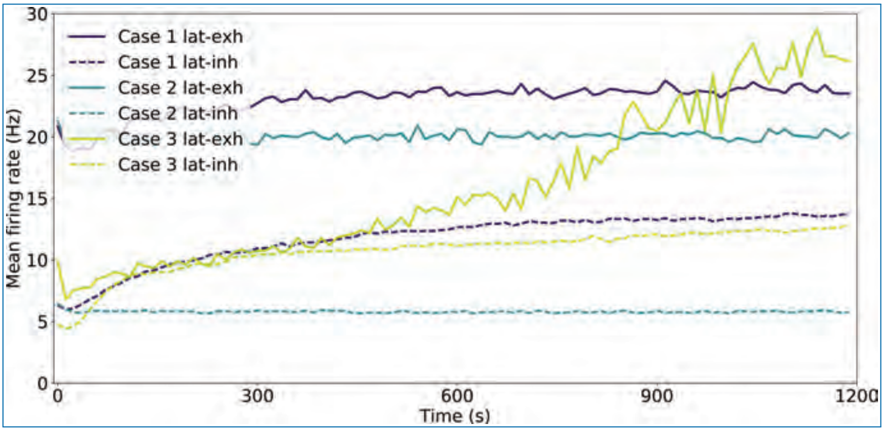


Figure 7.27. Target layer firing rate evolution throughout the simulation. The instantaneous firing rate has been computed in 1.2-second chunks for simulations where lateral connections are excitatory (lat-exh) and for simulations where lateral connections are inhibitory (lat-inh).

partner selection mechanism to focus its attention mostly on the target layer. We have achieved a reduction in the target layer firing rate by introducing inhibitory lateral connections. This is sufficient to generate a stable topographic mapping that matches quite closely the results of the original network when both input correlations and synaptic rewiring are present: $\sigma_{aff-conn} = 1.74$, $\sigma_{aff-weight} = 1.38$, $AD_{conn} = 0.85$, $AD_{weight} = 0.98$; all results are significant. The combined choice of sampling mechanism and lateral inhibition has a homeostatic effect upon the network.

Conversely, in the cases where input correlations are present, we see stable topographic mapping, regardless of the presence of synaptic rewiring, as well as significantly more feedforward synapses. Finally, no applicable network was negatively impacted by initialising the connectivity either randomly or with minimal connections.

To sum up, the model can generate transiently better topographic maps in the absence of correlated input when starting with a negligible number of initial connections or with completely random connectivity. These results can also be stabilised with the inclusion of lateral inhibitory connections preventing self-sustained waves of activity within the target layer. Experiments using correlated inputs do not require inhibitory lateral feedback either for reducing the spread of the receptive field or for maintaining a stable mapping. Finally, the model has proven it is sufficiently generic to accommodate changes in initial connectivity, as well as type of lateral connectivity, that is, the change from excitatory to inhibitory synapses.

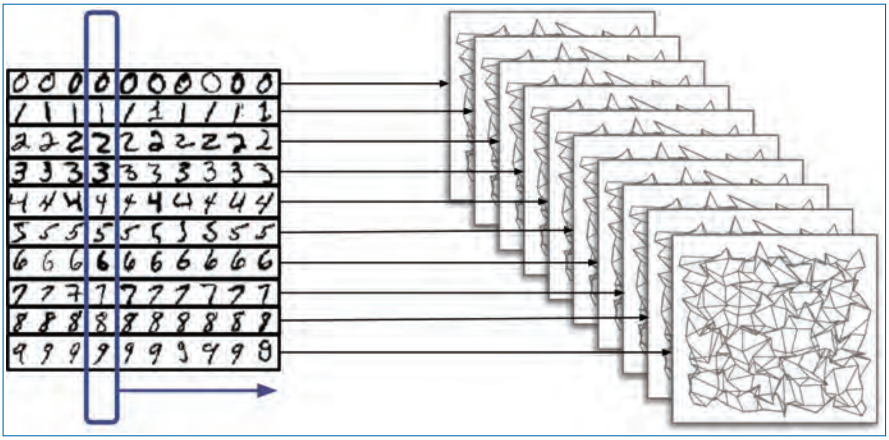


Figure 7.28. Network architecture used for training. A source layer displays a series of examples of handwritten digits; each example from a particular class is projected to the target layer corresponding to that class.

7.5.3 MNIST Classification in the Absence of Weight Changes

With an established model for synaptic rewiring [23], it is now possible to perform classification tasks using a simple architecture [103].

The model is equivalent to the supervised learning paradigm in ANNs. Data are labelled using a dedicated projection from a source layer to the corresponding target layer. A layer of neurons providing examples belonging to a class connects exclusively to a population which learns to recognise members of that class. Figure 7.28 shows the network architecture of the training regime, where each source in a source-target population pair displays a digit for 200 ms for a combined total simulation time of 300 seconds; the initial connectivity between each source-target pair is 1%.

To generate the input, each original digit is filtered via convolution using a 3 × 3 centre-surround kernel, mimicking the response of the highest resolution retinal ganglion cells. The kernel is normalised to sum to zero with an auto-correlation equal to one. Finally, a threshold is applied after the convolution operation, resulting in edge detection. Transmission within the network is achieved through the use of neurons generating Poisson spike trains; each pixel within the 28 × 28 image is mapped to two Poisson neurons, one for the on channel and one for the off channel. The top of Figure 7.29 shows examples of input digits before adding background noise; all of the feed-forward connections are excitatory.

The bottom image in Figure 7.29 shows that it is possible to identify visually what each target layer has learnt; time-averaged digits from each class are embedded

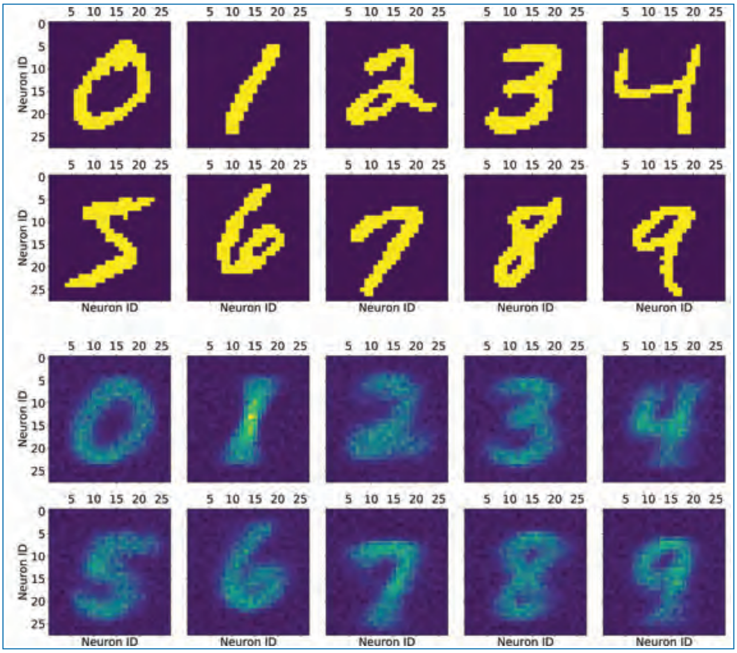


Figure 7.29. Top: Input rate-based MNIST digit representation. Bottom: Reconstruction of the learned digits when connectivity is adapted using only structural plasticity.

into the connectivity of the network. It is then possible to test the quality of classification. For this, we make use of a single source layer. The previously learnt connectivity is used to connect all of the target layers to the source layer, and all plasticity is disabled. The source layer now displays class-randomised examples, each for 200 ms. The classification decision is made off-line, based on which target layer has the highest average firing rate within the 200 ms period.

This is not a state-of-the-art MNIST classification network (it achieves a modest accuracy of 78% and a Root Mean Squared Error (RMSE) of 2.01, Figure 7.30 reveals what mistakes the network made) as each input digit class is represented only as an average for that class, but it serves here to demonstrate that synaptic rewiring can enable a network to learn, unsupervised, the statistics of its inputs. Moreover, with the current network and input configuration, the quality of the classification is critically dependent on the sampling mechanism employed in the formation of new synapses. Random rewiring, as opposed to preferentially forming connections to neurons that have spiked recently, could achieve accurate classification only if operating in conjunction with STDP.

Finally, this approach is also critically dependent on the encoding scheme used to represent the input. A rate-based encoding as shown here is required if no STDP is present.

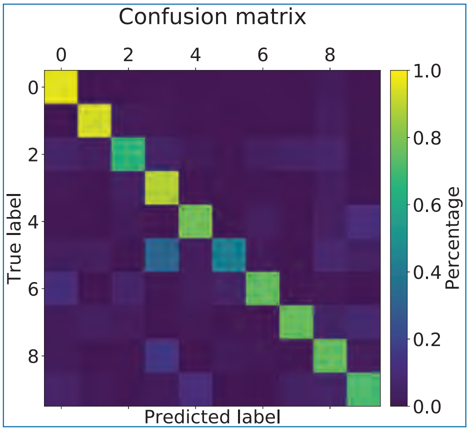


Figure 7.30. Classification confusion matrix.

7.5.4 Visualisation, Visualisation, Visualisation

While working on implementing such an alien mechanism on SpiNNaker, it was important to ensure that everything behaved correctly. It took more than a year to reach this goal. More than a year of simulations, trial and error, blood and sweat, printing parameters from SpiNNaker and saving them to disk, analysing and re-analysing. Frustratingly, we were stuck at debugging weird behaviour revealing itself as unexpected stripes in the connectivity matrix. Jupyter notebooks played a central role in this process, although in hindsight they should have been phased out after the initial first complete run. This particular issue had evaded all attempts at a systematic sweep of parameters and histogram plotting. Until one day…

Discipline is of course important in identifying bugs – this is why we can point at the exact simulation results that flagged the issue.[1] However, I believe that just as important as discipline can be a singular plot. In our case that plot is shown in the middle of Figure 7.31. It perfectly captures our issue: a systematic bias in the choice of pre-synaptic partner caused by the sub-millisecond ordering of spikes as they are generated by the input Poisson spike source.

The problem: we were selecting the last neuron to have fired as a partner for formation, thus ordering the spikes generated within a millisecond time step. The solution: correctly appreciate that because of the resolution of the time step, the input spikes were simultaneous and the source of one should be selected at random as a partner for rewiring.

To conclude, the solution only emerged after seeing a plot that would later be included in a paper by George *et al.* [73]. It looked at a vector of neurons that

[1] http://dx.doi.org/10.17632/xfp84r5hb7.1#folder-36833daa-91a8-499c-a898-65a96e22958b

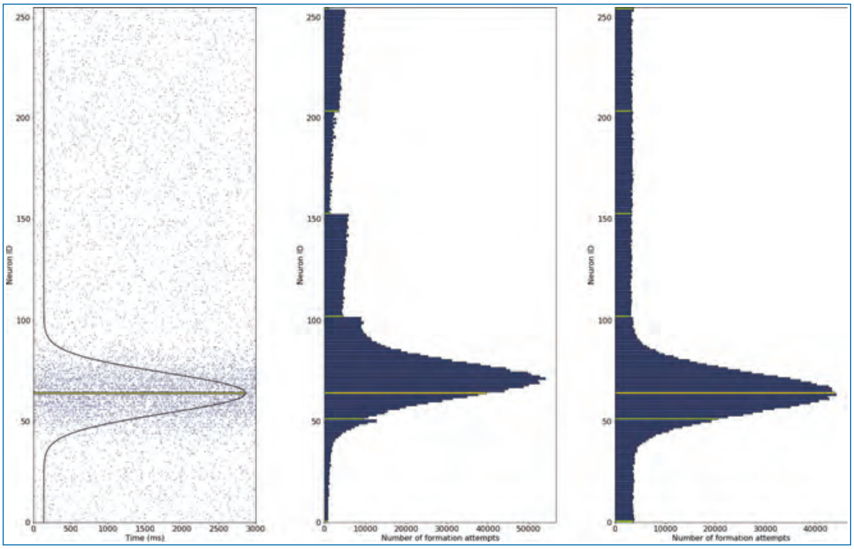


Figure 7.31. Left: the input spiking activity represented by a Poisson spike generator with rates described by the Gaussian curve in black; centre: the neuron identifiers considered for formation throughout an entire simulation if this choice relies on selecting the last neuron to have spiked; right: the neuron identifiers considered for formation throughout an entire simulation if the choice relies on selecting an arbitrary partner among the ones that have spiked since the last time step. In bright yellow, aligned across all plots: the neuron with the highest firing rate. Additionally, the partitioning of the pre-synaptic population is highlighted in green in the central and right-most figures.

generated input spikes with bimodal firing rates and showed how the connectivity had adjusted. We needed only look at which sources the formation attempts were considering to reveal the issue.

7.5.5 Rewiring for Motion Detection

We build on previous work [103] and present an end-to-end approach to perform elementary motion decomposition using LIF neurons and structural and synaptic plasticity [22]. Further, the computational platform which is the basis for these simulations is event-driven [207], including the spiking visual input provided to the network. The biologically inspired sensory processing method presented here is an alternative to traditional frame-based computer vision.

We show that (1) the presented architecture allows for unsupervised learning; that (2) synaptic rewiring enhanced to initialise synapses by drawing from a distribution of delays produces more specialised neurons; and that (3) a pair of readout neurons is sufficient to correctly classify the input based on the target layer's activity

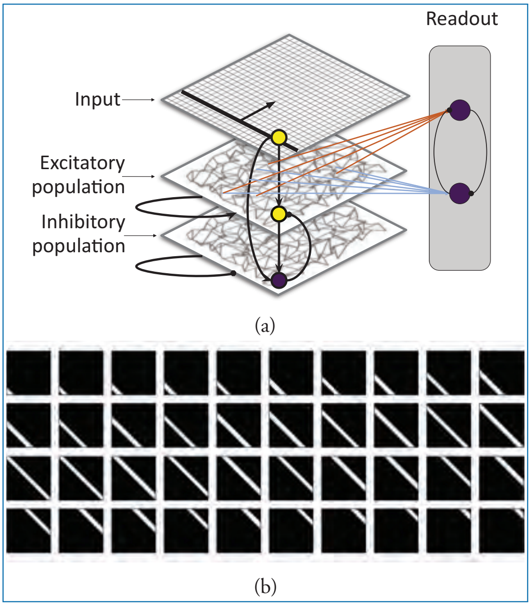


Figure 7.32. (a) Network architecture. (b) Example input 45° movement represented as its constituent frames (before processing to generate spikes). A new frame is presented every 5 ms and, in total, the presentation of an entire pattern takes 200 ms.

using rank-order encoding (first classification neuron to spike wins), rather than spike-rate encoding (classification neuron that fires most in a time period wins).

The SNN architecture (pictured in Figure 7.32(a)) is designed to allow unsupervised learning through self-organisation using synaptic and structural plasticity mechanisms [23]. Neurons in the two target populations are modelled as being positioned at integer locations on a 32 × 32 grid with periodic boundary conditions. The excitatory population contains neurons that receive sparse excitatory connections from the input layers and from themselves, while projecting to the inhibitory layer and to the readout neurons responsible for the final motion classification decision. The inhibitory population follows a similar structure, but only projects using inhibitory synapses. Very strong inhibition is also present between the readout neurons, implementing a WTA circuit. The networks are described using the PyNN simulator-independent language for building neuronal network models [44] and the SpiNNaker-specific software package for running PyNN simulations (sPyNNaker [207]).[2] The model is simulated in real time on the SpiNNaker many core neuromorphic platform using previously presented neuron and synapse dynamics [23].

[2] The data and code used to generate the results presented here are available from doi: 10.17632/wpzxh93vhx.1

The input stimulus consists of bars encoded using spikes representing 'ON' and 'OFF' pixels (see Figure 7.32(b) for an example before filtering using a previously described technique [194]) as well as a background level of Poisson noise (5 Hz). Each stimulus is presented over a 200 ms time period always moving at a constant speed (200 frames per second). During training, the target layers are presented with bars moving in two directions (Eastward or at 0° and Northward or at 90°), but during testing they are presented with moving bars in all directions (randomised over time, in 5° increments) – weights and connectivity are fixed during this latter phase. The simulations are initialised with no connections and are trained for around 5 hours, while testing occurs over 20 minutes. As a result of the chosen testing regime, the networks sees over 80 moving bar presentations at each of the 72 angles. This allows us to perform a *pair-wise independent t-test* between the responses at each of the angles in the two cases and establish whether their responses are statistically different. The readout neurons are trained and tested separately from the rest of the network – this process takes on the order of a minute.

Using the structural plasticity mechanism implemented for SpiNNaker, new synapses are formed in two regimes: with heterogeneous, random delays ([1, 15] ms, uniformly drawn) and homogeneous, constant (1 ms) delays; the latter is taken to be the control experiment. Further, according to the structural plasticity mechanism, depressed synapses are more likely to be removed. This optimises the use of the limited synaptic capacity available for each post-synaptic neuron [73]; neurons have a fixed maximum fan-in of 128 synapses with fixed delays.

The Direction Selectivity Index (DSI) will be computed for each neuron after training: $DSI = (R_{pref} - R_{null})/R_{pref}$, where R_{pref} is the response of a neuron in the preferred direction and R_{null} is the response in the opposite direction [157]. We compute it for each of the possible directions and establish the preferred direction as that which maximises the DSI. Individual neurons generally have noisy responses. As such, to avoid the noise skewing the computation, we filter the response of the neuron by applying a weighted average on individual angle responses.

More formal analysis, although less specific to the task of motion detection, is also performed. Entropy is computed using each neuron's normalised spiking profile. After testing, each neuron's firing profile $R(X)$ is computed in relation to each one of the $i = 1 \rightarrow 72$ input movement directions. Normalisation is performed by dividing every response by the sum of all responses:

$$P_i(X) = R_i(X) \sum_j R_j(X) \tag{7.31}$$

so that $\sum_i P_i = 1$ for each neuron X. The firing profile of the neuron can thus be interpreted as the neuron's confidence in the input movement direction. We can compute the entropy of a neuron's response:

$$H(X) = -\sum_{i} P_i(X) \log_2 P_i(X) \tag{7.32}$$

The maximum entropy in the presented system is thus $-\log_2(1/72) \approx 6.17$ bits, which is equivalent to neurons displaying equal spiking activity in all presented angles, or no selectivity whatsoever. Neuron X is said to be very selective if simultaneously maximises DSI ($DSI(X) \rightarrow 1$) and minimises entropy ($H(X) \rightarrow 0$). It is sufficient for a neuron to have $DSI(X) \geq 0.5$ to be considered selective.

Both DSI and entropy are used to select and investigate the behaviour of individual neurons. In the following section, we will display quadruplets of individual neuron responses that have maximal grid-aligned responses and minimal orthogonal and opposite responses: $argmax_X = R_i - (R_{i+90^\circ} + R_{i-90^\circ} + R_{i+180^\circ})$. Here i is in turn: 0°, 90°, 180° and 270°, that is, the cardinal directions. The distributions of entropy and DSI are also included for all experiments. Comparison of these distributions is performed by applying both Welch's t-test and Kruskal's h-test.

After training the readout units, it is possible to establish class inputs. Based on the predicted label and the known true labels, we report the accuracy, recall F-score of the network, as well as the RMSE. We define Tp and Tn to mean the number of true positive and true negative examples, while Fp and Fn refer to the number of false positives and negatives, respectively. Recall or sensitivity, intuitively the ability of the classifier to find positive samples, is reported as $Tp/(Tp+Fn)$. Precision or the positive predictive value, intuitively the qualitative ability of the classifier not to label as positive an example that is negative, is computed as $Tp/(Tp+Fp)$. Given these metrics, we can now compute the weighted average between precision and recall to generate the F-score $F1 = 2$ (precision × recall)/(precision + recall). Due to the readout architecture and experimental parameters, we also investigate the number of instances in which no readout neuron produces a spike.

The response of the excitatory population in each regime (incorporating heterogeneous delays or not) is plotted for each testing direction (minimum, mean and maximum responses presented in Figure 7.33(a)). The polar plot reveals the firing rate (Hz) of neurons during testing when the input is moving in each of the 72 directions from 0° to 355° in 5° increments in a random order. The network response shows that neurons are responding preferentially to movement, rather than simply to the shape of the input, because the response is asymmetrical – it can differentiate between, for example, a vertical bar moving eastward and the same vertical bar moving westward. The *pair-wise independent t-test* is performed to compare the network response in the two regimes (Figure 7.33(c), red line signifies that $p \geq 0.001$ for that particular angle); the response is higher in one training direction

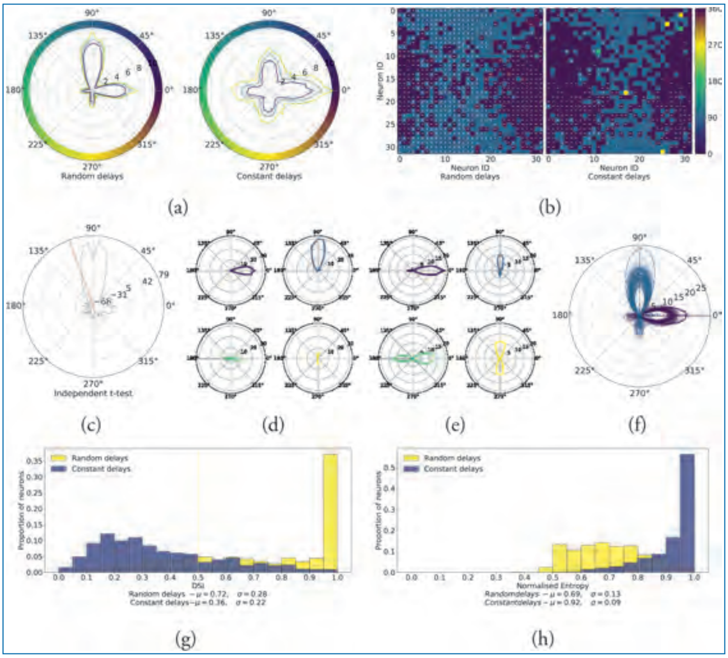
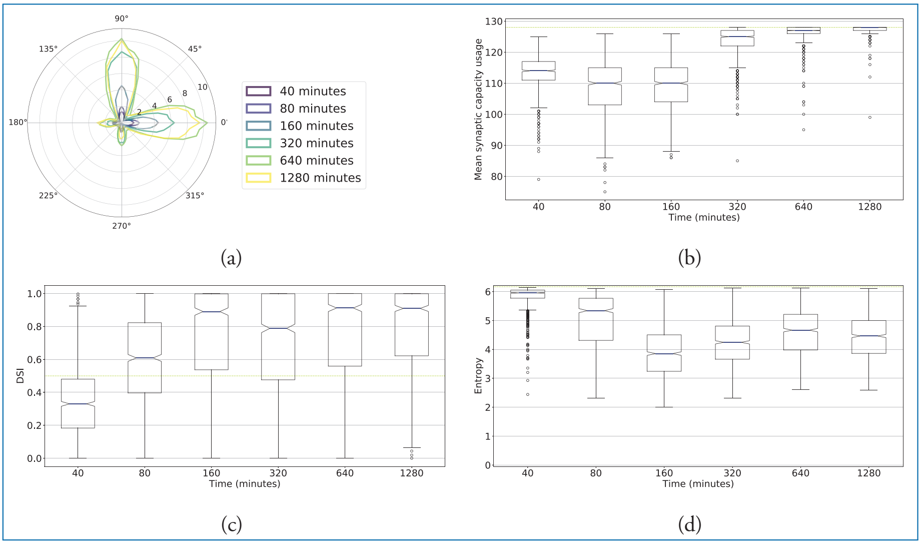


Figure 7.33. Spiking activity comparison between networks where rewiring assigns random and constant delays, respectively, to new synapses. Both networks were trained using bars moving eastward (0°) and northward (90°). (a) the minimum, mean and maximum aggregate excitatory population firing response (Hz); (b) neuron angle preference based on maximum firing rate encoded by the colour and DSI represented by the arrow direction (it is only present if DSI ≥0.5); (c) pair-wise independent *t*-test comparing the network with heterogeneous delays (on the left in a and b) to the control setup (on the right in the same subplots) – red lines show the angle at which the comparison yielded insignificant results; (d) selected individual neuron responses in the 4 grid-aligned directions (random delays); (e) selected individual neuron responses in the 4 grid-aligned directions (constant delays); (f) individual overlaid selective neuron responses after filtering (DSI ≥0.5); (g) histogram comparison of all DSI values in the two networks; (h) histogram comparison of all entropy values in the two networks.

(90°) and less in the other (0°) for the network with heterogeneous delays compared to the control. As such, we proceed by examining individual neurons rather than the average network behaviour. The spatial organisation of neurons and their preferred angle is presented in Figure 7.33(b), showing that local neural neighbourhoods become sensitised to the same input statistics. There we also look at neurons' maximum responses (encoded by the colour of the cell) in conjunction with the direction that maximises DSI (arrow direction) and DSI ≥ 0.5 (arrow presence). The DSI histogram presented in Figure 7.33(g) compares the two networks; the



(a)

(b)

(c)

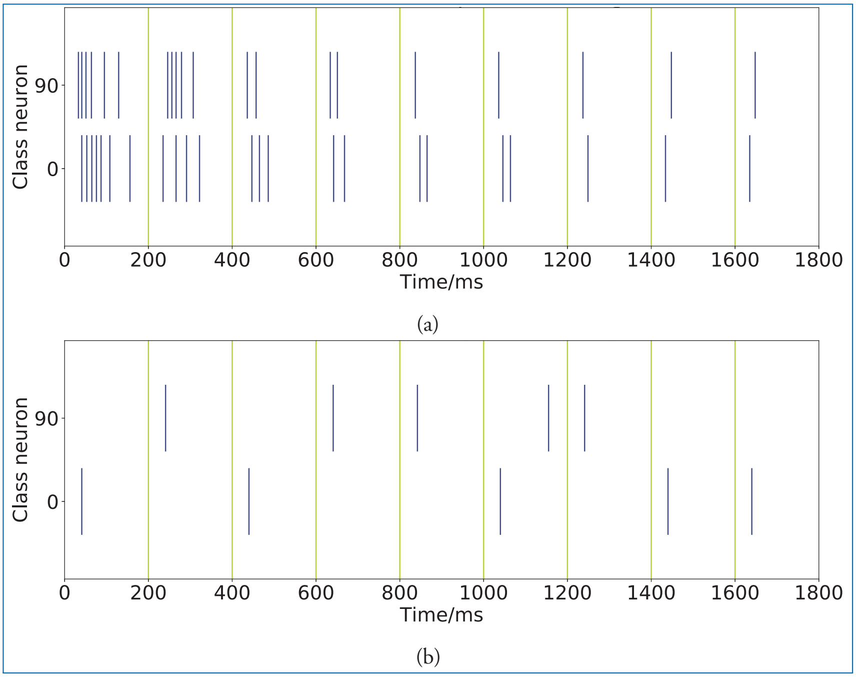
(d)

Figure 7.34. Network evolution over a wide range of simulation run times when trained on two angles. (a) average network firing response during inference when trained for ever increasing times; (b) average number of afferents (incoming connections) for each neuron in the excitatory target layer; (c) DSI distribution displayed as a *boxplot* for each simulation in (a); (d) entropy distribution displayed as a *boxplot* for each simulation in (a). Note: Each data point is a different simulation.

control network has significantly fewer selective neurons (251 compared to 744) and selectivity is lower on average. Individual responses of our simulated neurons resemble the direction selectivity found in Superior Colliculus [112].

Further, we examine the network behaviour over a wide range of simulations times, ranging from 40 minutes up to 20 hours. Figure 7.34(a) shows the evolution of the population-level firing rate and the evolution of the DSI metric (Figure 7.34(c)). The network is thus shown to be stable over long periods of time, rather than showing destructive dynamics.

A readout or classification mechanism relying on two mutually inhibitory neurons is sufficient to resolve the two directions presented in the input. Static excitatory connectivity originating from the excitatory layer results in a potential 100% classification accuracy based on rank-order encoding. After 40 seconds, the two neurons have self-organised to respond to one of two input patterns. Figure 7.35 shows the spiking behaviour of the two neurons in the first 1.8 seconds of training and testing. STDP reduces the latency in neural response to the stimuli, making the neurons respond to the stimulus onset, thus making them ideal for classification using rank-order encoding, rather than a WTA classification based on spike count across a time period [103].



(a)

(b)

Figure 7.35. Initial spiking activity of the two readout neurons during training (a) and testing (b). The full-height vertical bars denote the edges of the pattern presentation time bins (every $t_{stim} = 200$ ms). Neuron class is established *post hoc* as the one maximising classification accuracy.

7.6 Neuroevolution

SNNs are not solely defined by hierarchical-layered architectures and activation functions but also by neuron model parameters that can alter neuron behaviour over time. The extra spatio-temporal degrees of freedom afforded by SNNs give models a larger parameter space than ANNs. This is of particular note for SNNs that are models of biological neural networks where experimental or ethical limitations mean that it can be difficult or impossible to collect the data *in vivo* to define all the necessary model parameters. Optimisation techniques can be used to explore the parameter spaces of such models and find plausible values. We anticipate that this will be a fruitful area of future research.

A practical method for optimisation and the exploration of the large search spaces seen with SNNs is to use gradient-free (also known as random search) optimisation methods such as Evolutionary Algorithms (EAs). EAs involve evaluating a population of potential solutions (known as agents or individuals) against a

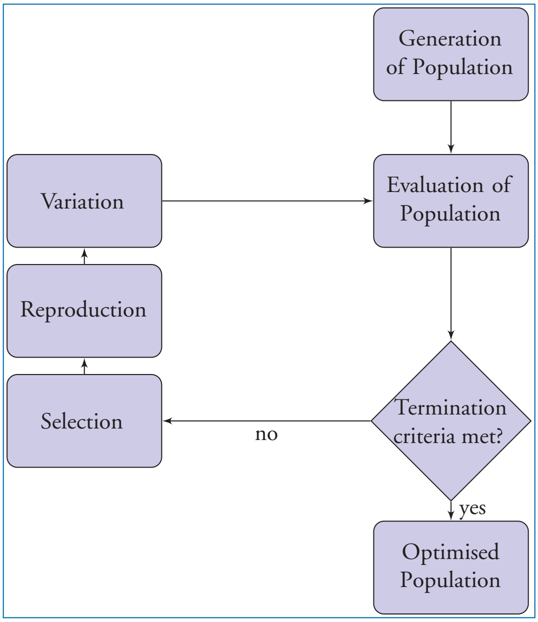


Figure 7.36. A flowchart showing the steps in an EA.

target task.[3] The agents in the population that perform best are selected to reproduce, with the offspring having some variation applied. The offspring form the population for the next generation, see Figure 7.36, often with a small portion of the best-performing agents automatically passing into the next generation. This is known as elitism and has been shown to aid convergence to solutions [46]. Like evolution by natural selection in biology [43], survival of the fittest in EAs is an unguided force which can lead towards better performing agents.

How the performance of agents is evaluated is task specific; however, it does not require gradient information, making an EA approach particularly well suited for tasks in which the error is either difficult or impossible to differentiate. The terms EA and Genetic Algorithm (GA) are often used interchangeably but technically GAs are a subclass of EAs in which agents are encoded as discrete values in 'genes' and random mutation and crossover adding variation to offspring.

The scale of large SpiNNaker systems, such as the one million core machine at the University of Manchester, lend themselves to population-based search methods. The parallelism available with such systems allows model execution to become invariant with respect to population or network size, the main components leading

[3] So far, population has been used in the context of PyNN to describe a group of neurons. In the context of EAs population is used to refer to a group of individuals (whole networks) to be optimised, unless otherwise specified.

to increased simulation time on other platforms. Larger population sizes increase the effectiveness of an EA as they increase the coverage of the search space. With more serial systems, a practical limit is placed on the size of the population but with SpiNNaker such scaling problems are diminished. There is an overhead in terms of loading models on to SpiNNaker; however, by separating the population of networks into a collection of smaller jobs, the loading time can be made similar to the time taken to run a single network assuming sufficient auxiliary computational resource.

7.6.1 Pac-Man on SpiNNaker

The first concrete application of neuroevolution to develop SNNs on SpiNNaker was undertaken by Vandesompele *et al.* [259] in which the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [237] was used to generate an agent able to play the Pac-Man arcade game as well as solve the XOR problem. In Pac-Man, the agent had a view of the tiles one or two spaces around it, closer to modelling an agent in a maze than a conventional human player. The algorithm converges on the XOR problem, although it required 49,750 evaluations compared to the 13,459 in the worst-case performance of NEAT using an ANN (average of 4,800). The Pac-Man game was a proof of concept. The performance did improve with time; however, it did not get close to the theoretical maximum score. This is to be expected of an agent that cannot see further than two positions around itself as any high level of forward planning necessary to eat ghosts, representing a large increase in score, is not feasible.

7.6.2 Further Exploration of NEAT

Work is currently being undertaken to increase the efficiency of the algorithmic implementation of NEAT on SpiNNaker. One key aspect to improving performance is to run the target task on SpiNNaker directly so that the model and the environment with which it interacts need not communicate via Ethernet; instead of having spike communication with a host computer, connections are kept local and fast, eliminating an input/output bottleneck and increasing the scale of population possible with SpiNNaker. So far, models running on SpiNNaker have been created for the multi-armed bandit task, inverted pendulum and Breakout (the classic Atari game) as well as a number of other tasks.

7.6.3 An Evolutionary Optimisation Framework for SpiNNaker

Another approach developed an EA framework for SNN models running on SpiNNaker and looked to understand the scaling and performance of this kind of optimisation. The weight parameters of a small convolutional network for the MNIST digit recognition task were evolved as a test case. A GA was chosen as the optimisation algorithm because of its biological relevance and the potential for model evaluation to be parallelised by running multiple models simultaneously on SpiNNaker.

7.6.4 Methods

Figure 7.37 shows the structure of the simple convolutional SNN model, the weights of which were optimised for the MNIST digit recognition task using a GA. The spikes of the 28 × 28 input layer were rate-coded representations generated from the MNIST images. The hidden layer was a 24 × 24 layer that is the convolution of input with a 5 × 5 filter. The hidden layer was fully connected to the 10 output neurons. The 25 weights of the filter and the 5,760 weights of the fully connected layer were encoded in a 5,785 base gene, with the bases taking integer values in the range −1 to +1. The details of the GA used are detailed in Table 7.9. Two experiments were carried out to better understand the effect of different initialisation on the evolution of the population over 304 generations: the

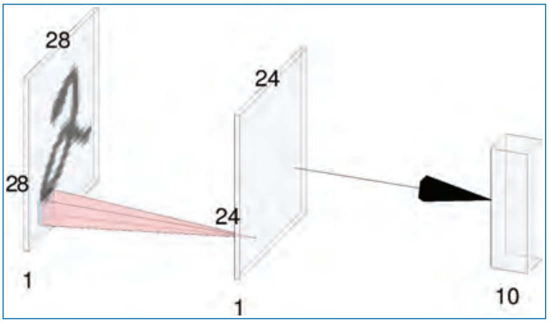


Figure 7.37. The structure of the simple SNN model optimised using a GA.

Table 7.9. A summary of the GA parameters.

Variable	Value
Population size (individuals)	24,000
Mutation rate	0.1%
Mutation type	Base substitution
Crossover rate	50%
Crossover type	Two-point crossover

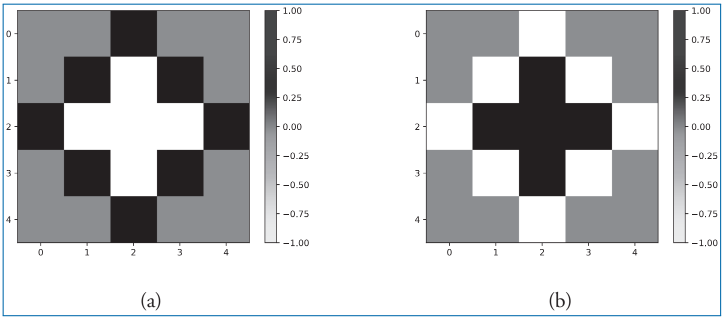


Figure 7.38. The centre-surround filters used to seed the seeded population.

first experiment tracked the accuracy of a population of 24,000 individuals with randomly initialised genes (the unseeded population) and in the second the initial population (the seeded population) was seeded with 12,000 individuals with one centre-surround filter (6,000 positive, 6,000 negative, see Figure 7.38). In each of these experiments, 7.29×10^6 networks were evaluated on SpiNNaker.

7.6.5 Results

Figure 7.39 shows the evolution of the training accuracy of the two populations over 304 generations. The five top performing individuals from the final populations were evaluated against the MNIST testing set and the best individuals gave 66.7% and 63.9% testing accuracy, unseeded and seeded, respectively. These results are far from state-of-the-art accuracies but demonstrate that a GA can be used for optimisation in this way.

It was hypothesised that the same convolutional filter may evolve from an unseeded population independent of the random initialisation; however, multiple runs of smaller populations showed that, on the order of hundreds of generations, this is not the case. During the course of these experiments, the time performance of the system was evaluated. It was found that the overhead of submitting a Spalloc job merited redesigning the framework to allow multiple models to be evaluated in one job. As a reminder, Spalloc is the current SpiNNaker job submission system which allocates a subset of the entire machine to individual users. This work demonstrated that it is possible and feasible to use a GA to tune the parameters of an SNN model on SpiNNaker.

7.6.6 Future Work

There are a number of avenues for future work in the field of SpiNNaker and optimisation algorithms. Future work could involve the use of different optimisation algorithms, and the application of optimisation algorithms to the conversion of

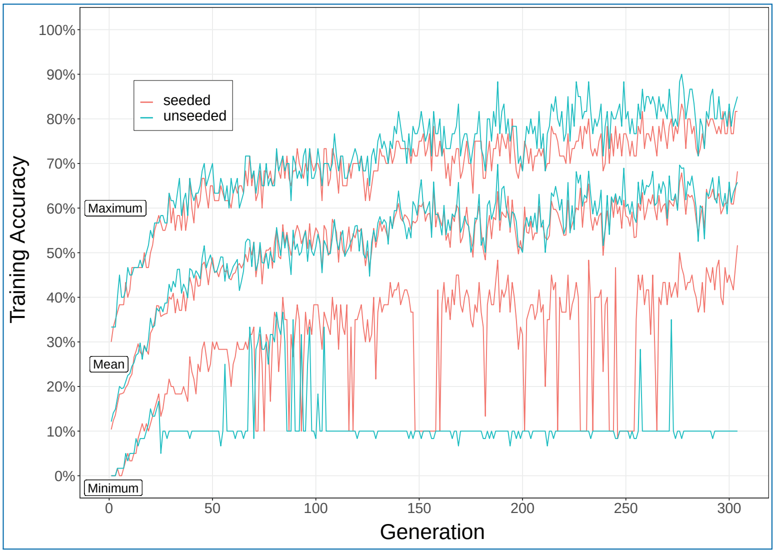


Figure 7.39. A graph comparing number of generations compared with training accuracy for a GA with two populations of 24,000 individuals, each being a SNN model. The seeded population was initialised with 12,000 centre-surround filters, 6,000 positive and 6,000 negative and 12,000 individuals with randomly initialised filters. The unseeded population was initialised with 24,000 random individuals.

ANN models and to uncover novel learning mechanisms in SNNs. Looking more broadly, the development of robust model optimisation frameworks could well lead to a change in the way that research is carried out.

Different EAs

The modification of genes in EAs is random and mutations are undirected; an Evolutionary Strategies (ES) algorithm [15] may well be able to achieve similar results with far fewer evaluations. It does this by mutating the best agent in a population multiple times to create a new generation. The mutation vector of all individuals is then scaled by their performance and totalled to give an approximation of the gradient allowing the next-generation mutation to be in the direction in the solution space which produced the best performance.

One of the key features of SpiNNaker is its asynchronicity, a feature that lends itself to lesser-studied asynchronous steady-state EAs [2]. In these types of models, fitness values are not gathered at the end of a generation as they would be in a typical generational GA, rather models compete between themselves locally. This kind of optimisation algorithm could be run directly on SpiNNaker and would minimise the overheads associated with scattering and gathering data to across multiple chips and boards. We estimate that running the EA 'on-machine' could half the time taken to evaluate models similar in scale to that of the MNIST test network above.

Machine Learning

A key area of interest is understanding the relationship between SNN and ANN models and translation between them. Deep ANN models trained by error backpropagation give state-of-the-art performance in many benchmark machine learning tasks. An automated optimisation framework could be used to help convert ANN models to SNNs to be run on SpiNNaker. This would dramatically increase energy efficiency for practical applications as well as adding to our understanding of how information is processed in neural networks more generally.

Learning-to-Learn

A current fruitful area of research, dubbed Learning-to-Learn (L2L), applies optimisation techniques to fine tune the hyperparameters of another optimisation algorithm [12]. An example of this could be using a genetic algorithm to evolve the parameters which control how backpropagation performs. This can help find certain parameters and starting conditions conducive of fast learning on novel tasks. With both optimisers working at different time scales, it simulates the slow evolutionary process of genetic variation with the fast time scale acting in a similar way to learned behaviour during a lifetime.

Impact on Computational Neuroscience

By allowing work with biological models that do not have full parameter sets, the focus of researchers working with experimentally derived models could move towards higher levels of abstraction and larger-scale models. This step is an important one to bridge the gap between understanding information processing in the brain and the application of such knowledge in future neuromorphic systems.

DOI: 10.1561/9781680836530.ch8

Chapter 8

Creating the Future

By Dave Clark, Andreas Dixius, Steve Furber, Jim Garside, Michael Hopkins, Sebastian Höppner, Dongwei Hu, Florian Kelber, Gengting Liu, Christian Mayr, Mantas Mikaitis, Felix Neumärker, Johannes Partzsch, Stefan Schiefer, Stefan Scholze, Delong Shang and Marco Stolba

> *The best way to predict the future is to create it.*
>
> — Abraham Lincoln

In this chapter, we take a look into the future of this technology. First we survey interesting developments in hardware accelerators for SNNs and ANNs, but then we focus primarily on the second-generation SpiNNaker developments. Here we will refer to the current SpiNNaker machine as SpiNNaker1 and the second-generation machine as SpiNNaker2.

8.1 Survey of Currently Available Accelerators

This is an exciting time as large corporations are exploring the usefulness of neuromorphic systems, and it is no longer an effort driven solely by academia. Most systems are in a research prototype phase, rather than fully commercially viable products. Current offerings include:

- Intel's Loihi[1]; IBM's TrueNorth[2]; Eta Compute's Tensai[3]; aiCTX[4]; BrainChip's Akida.[5]

Although neuromorphic systems are still at an early stage, many companies already offer or are close to offering options for accelerating inference and learning in artificial DNNs. These include:

- In data centres: NVidia's Tesla Graphics Processing Units (GPUs) (P100 for training, P4 & P40 for inference)[6]; Intel's Nervana L-1000 Neural Network Processor (NNP)[7]; Graphcore's Colossus Intelligence Processing Unit (IPU)[8]; Google's Tensor Processing Unit (TPU).[9]
- In mobile devices: Huawei's Kirin970 AI Processor[10]; Qualcomm's AI Engine[11]; Imagination's PowerVR Series2NX and Series3NX[12]; Apple's A12 Bionic[13]; Cadence's and Tensilica's HiFi 5 Digital Signal Processor (DSP)[14]; ARM's Trillium Project[15]; LG's AI Chip.[16]

[1] https://newsroom.intel.com/tag/loihi/#gs.7e79qw

[2] http://www.research.ibm.com/articles/brain-chip.shtml

[3] https://etacompute.com/

[4] https://aictx.ai/

[5] https://www.brainchipinc.com/

[6] https://www.nvidia.com/en-gb/deep-learning-ai/

[7] https://www.intel.ai/nervana-nnp/

[8] https://www.graphcore.ai/

[9] https://cloud.google.com/tpu/

[10] http://www.hisilicon.com/en/Media-Center/News/Key-Information-About-the-Huawei-Kirin970

[11] https://www.qualcomm.com/snapdragon/artificial-intelligence

[12] https://www.imgtec.com/vision-ai/powervr-series3nx/

[13] https://www.apple.com/uk/iphone-xs/a12-bionic/

[14] http://www.cadence.com/go/hifi5

[15] https://www.arm.com/products/silicon-ip-cpu/machine-learning/project-trillium

[16] http://www.lgnewsroom.com/2019/05/lg-to-accelerate-development-of-artificial-intelligence-with-own-ai-chip-2/

Applications of these accelerators mostly focus on visual pattern recognition and inference, that is, ConvNets, and thus, they share some common features. Firstly, data movement incurs time and energy costs that are minimised by keeping the data close to the computational unit, a trait that SpiNNaker shares. Secondly, at least during inference, high accuracy can be achieved with relatively low precision computation.

Regardless of the level of biological plausibility, this explosion of products and interest in custom accelerators seems to be driven by a desire for cars to drive themselves and for mobile phones to perform inference locally rather than relying on data centres as at present.

This section captured a snapshot of the current state of hardware development for accelerating neural network processing. At the time of reading, it is most likely out of date. However, its purpose is to show the very high interest both in neural networks and in designing ASICs to compute efficiently in spite of Moore's law [168] coming to its end.

8.2 SpiNNaker2

In this increasingly competitive environment of novel ANN and SNN hardware offerings, we embarked upon the development of a second-generation SpiNNaker system. The resulting chip will be fabricated in 2020, and small- and large-scale systems will follow. Here we present the thinking behind SpiNNaker2, and the approach taken to its design.

8.2.1 Lessons from SpiNNaker1

The full SpiNNaker1 chip was delivered in 2011, since when considerable experience has been gained in its use. This experience allows us to identify the strengths and weaknesses of the design, and in the second-generation system we can build on the former and try to address the latter.

Strengths

- *Software neuron and synapse modelling*. Although the use of software inevitably compromises energy-efficiency compared with hard-wired analogue or digital algorithms, for a *research* platform we believe that the resulting flexibility more than warrants this sacrifice.
- *Multicast packet routeing*. The SpiNNaker packet routeing mechanism has proved its ability to adapt to a wide range of use profiles and

is still the most flexible neuromorphic interconnect mechanism devised to date.

Weaknesses

- *Host I/O performance*. The 100 Mbit Ethernet I/O on each SpiNNaker1 board has proved a major bottleneck in the machine's use. Although we have found ways to circumvent this bottleneck in a number of circumstances, much higher I/O bandwidth would greatly improve the performance and usability of the machine.
- *Memory sharing*. In SpiNNaker1, each processor core has its private local memory and (slower) access to the shared SDRAM. The trend towards increased communication between the cores on a chip, for example, when neuron and synapse modelling runs on different cores, has made moving data between cores increasingly important, but on SpiNNaker1 this can only be done via SDRAM.

In addition to these strengths and weaknesses, it is notable that the SpiNNaker1 cores spend a lot of time in a few oft-repeated algorithms for functions such as random number generation and computing exponentials. All of this experience has been fed into the design of the second-generation system, SpiNNaker2.

8.2.2 Scaling Performance and Efficiency

An obvious way to scale the performance and improve the energy efficiency of the next-generation SpiNNaker is to benefit from Moore's Law [168] and implement this multi-processor SoC (MPSoC) in an advanced technology node. Here the 22-nm Fully-depleted Silicon-on-Insulator (FDSOI) technology 22FDX by GlobalFoundries [34] has been chosen.

The second aspect for improvements is the application of circuit design techniques to enhance both the compute performance and the energy efficiency of SpiNNaker2. This includes specific hardware extensions to the neuromorphic Processing Element (PE) as shown in Section 8.5 and power management techniques, such as dynamic voltage and frequency scaling [105], that have been proven on previous MPSoCs in the fields of mobile communication [88, 181] and database processing [87].

The overall target for the SpiNNaker2 system is to enhance the capacity for brain-size spiking network simulation in biological real time by at least 10×, at the same power consumption as SpiNNaker1. The envisioned scaling involves the three key aspects for neuromorphic computing of the processing of neural states and synaptic weights, the communication of spike events and the storage of synaptic weights.

8.3 SpiNNaker2 Chip Architecture

Figure 8.1 shows the SpiNNaker2 chip top-level architecture. It follows the same concept as the SpiNNaker1 neuromorphic computation system with:

- a homogeneous array of ARM core-based processing elements for software-defined neuromorphic computation;
- a lightweight spike packet communication fabric based on the SpiNNaker router (see Section 2.2.3) and energy efficient chip-to-chip links;
- off-chip SDRAM interfaced by two Low-Power Double Data Rate version 4 (LPDDR4) memory interface channels for synaptic weight storage.

The processing elements (PEs) are arranged in groups of quads (QPEs) which form tiles for the homogeneous processor array. SpiNNaker2 employs a mesh-based NoC where every QPE constitutes one node of the mesh grid. The NoC is responsible for all types of communication between the on-chip components and from/to the off-chip interfaces. This includes chip boot-up and configuration data transfers,

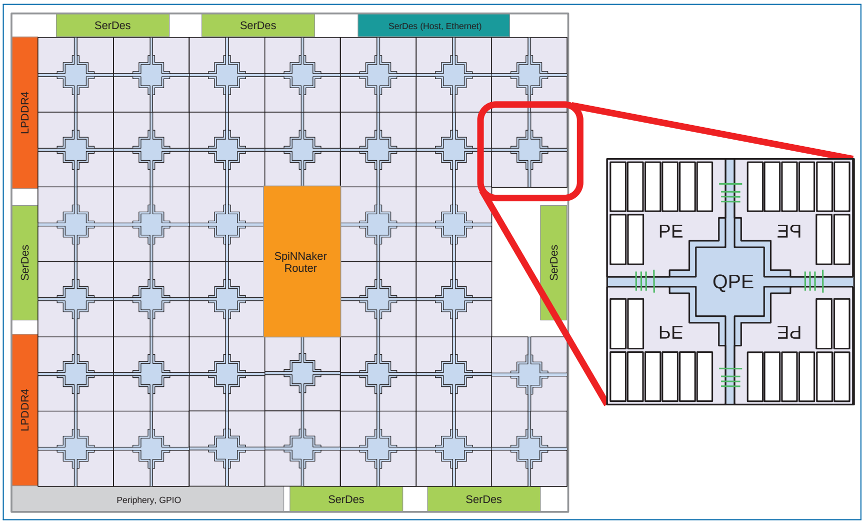


Figure 8.1. SpiNNaker2 chip architecture. The chip mainly comprises a 7 × 6 array of Quad Processing Elements (QPEs), with two of these replaced by the SpiNNaker router and two by the east inter-chip Serialiser/Deserialiser (SerDes) link, leaving 38 QPEs incorporating in total 152 PEs.

spike traffic and off-chip memory data traffic. Therefore, all other chip top-level components are connected to the NoC as well. These include the following:

- Two LPDDR4 memory interfaces, consisting of a memory controller and a PHY for a total maximum external memory bandwidth of 6 GBytes/s.
- The SpiNNaker2 router, which is similar in principle to the SpiNNaker1 router but with a larger routeing table (16 k vs. 1 k entries) and a number of other optimisations.
- Six bidirectional chip-to-chip links with up to 3 GByte/s per link in each direction. These links employ features such as low I/O signal voltage swing and forwarded clocking to achieve low power consumption, proportional to the data payload traffic.
- The host interface that provides connectivity to an external Ethernet PHY by means of SGMII with 4 lanes with up to 2.5 Gbits/s each.
- A periphery block providing functionality for autonomous chip boot-up, housekeeping and standard interfaces to off-chip sensors and actors for neuromorphic applications.

The definition of the SpiNNaker2 chip architecture is optimised for efficient implementation and verification. This includes the following aspects:

- The strictly homogeneous architecture partitioning of PE and QPE allows for hierarchical physical design implementation. This significantly reduces the tool run-times for synthesis and place and route to enable layout implementation within reasonable time frames.
- The QPEs and all other macro blocks are realised for connection to their neighbour directly by abutment placing. There are no flat top-level routeing signals in the SpiNNaker2 core area.
- Signal interfaces between macro blocks are asynchronously decoupled by FIFOs, realising a Globally Asynchronous Locally Synchronous (GALS) architecture. This avoids the need to implement a globally synchronous clock distribution network (e.g. a clock tree) and allows for individual frequency scaling of the individual components.
- The packet-based NoC communication fabric handles various types of data traffic, including configuration data, Design for Test (DfT) scan data, interrupts, spikes, DMA traffic. This avoids the need for various side signals, such as interrupt signals or DfT signal nets, which would complicate the top-level physical implementation.
- The DfT concept is strictly hierarchical, with scan chains inserted per macro block. The PEs can be set to DfT mode during system operation and execute tests (stuck-at scan, MBIST) with data transfer over the NoC.

8.4 SpiNNaker2 Packet Router

The SpiNNaker router is the key component in the SpiNNaker machine for SNN simulations. The SpiNNaker1 router was described in detail in Section 2.2.3. The router on the second-generation SpiNNaker chip incorporates improvements required here to support the larger routeing tables and increased communication throughput; the SpiNNaker2 chip contains more than 100 processing elements. All the on-chip and inter-chip spikes are routed by the SpiNNaker router. The whole SpiNNaker2 packet router is designed with fully pipelined packet flows and provides higher throughput and performance. The top-level structure of the SpiNNaker2 packet router is shown in Figure 8.2.

The SpiNNaker2 packet router has 6 (parameterised) on-chip and 7 off-chip communication channels occupying the same area as 2 QPEs. These channels attach and share the 6 bi-directional NoC ports running at a 400 MHz speed. Compared with the single input stream in the SpiNNaker1 packet router (running at around 100 MHz), the 6 parallel ports can absorb 2.4 G input packets per second (GPKT/s) which is 24 times larger than the SpiNNaker1 packet router. Furthermore, the parallel routeing engines enable the SpiNNaker2 packet router to have better routeing efficiency than the SpiNNaker1 packet router where only one packet can be processed every cycle at maximum.

The SpiNNaker2 packet router is currently designed to run at 400 MHz (via 6 NoC ports) which maintains the same maximum theoretical throughput as the SpiNNaker1 packet router. However, the realistic throughput will be increased by improving the communication bottleneck. The PE running at the same speed (200 MHz) now can take packet from the network every 1 or 2 processor cycles. The bandwidth of the off-chip I/Os is also increased significantly.

The output star network of the SpiNNaker1 packet router is an efficient network for multicasting. However, the centralised arbitration and buffering limit its scalability. The SpiNNaker2 chip will incorporate 152 PEs. Therefore, a 2D mesh network is chosen to provide a better scalability. Compared with SpiNNaker1, the different network topology also brings different design challenges for the SpiNNaker2 packet router.

The basic function of a router is to route each packet to its destination(s). However, the SpiNNaker2 packet router is more complicated than that, performing different routeing algorithms, system monitoring functions and including power optimisation and high-performance circuits within a limited power and area budget. Below are some new features and differences compared with the SpiNNaker1 packet router.

The 7th SpiNNaker link: This is an additional SpiNNaker link which is functionally similar to the other 6 SpiNNaker links. There are several advantages of

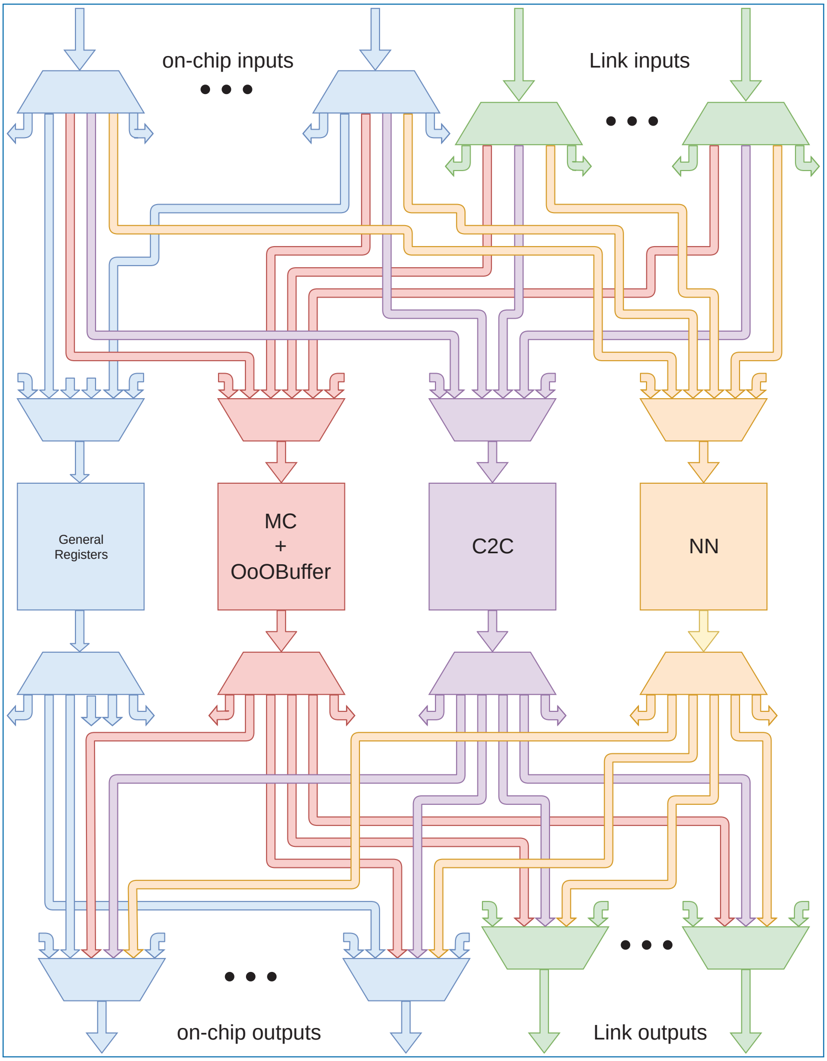


Figure 8.2. The internal organisation of the SpiNNaker2 packet router.

this additional link. First, the addition of the 7th link provides a dedicated connection for the interaction with other neuromorphic devices without breaking the torus which is already formed using the other 6 SpiNNaker link. Second, the 7th link can be used for a hyperconnection to another node in the system which can significantly reduce the routeing latency in the simulation, because the routeing through this short-cut path does not need to pass through multiple nodes to reach the destination. The disadvantage is that it introduces cost where the size of the multicast look-up table increases by adding one more destination bit per entry. However, it does not incur extra cost to the SpiNNaker core-to-core (C2C) and nearest-neighbour (NN) routeing, because the 3-bit destination information in the C2C look-up table and the 3-bit outgoing path information in the NN packets are not fully occupied in SpiNNaker1.

Interrupt packet messaging mechanism: The SpiNNaker1 packet router has direct interrupt lines to all the processing elements where the processing elements have interrupt vector controllers deciding whether or not to pay attention to the interrupt. This dedicated interrupt network is expensive to implement in SpiNNaker2 due to the cost and timing closure problem in a large area of silicon. All the communications between the SpiNNaker2 packet router and other units are designed to be carried by the on-chip data network (DNoC). Therefore, an interrupt messaging mechanism, sending exception packets, has been devised.

Out-of-order issue buffer: The output *star* network in the SpiNNaker1 packet router can issue a single multicast (MC) packet efficiently. The output strategy for MC packets is *All or Nothing* (AoN) where the MC packet will only be sent if all of its destinations are available. Therefore, in the SpiNNaker1 packet router, if one MC packet stalls due to an unavailable destination, all the subsequent MC packets will stall. In the SpiNNaker2 packet router, the out-of-order issue buffer is designed to further improve the output efficiency of MC packets. If the first MC packet stalls at the output of the multicast routeing engine, it will move to the out-of-order buffer unit. The out-of-order issue buffer can accommodate several MC packets. At each router clock cycle, the out-of-order issue buffer can send any MC packet which does not have a blocked destination. The output efficiency is increased by issuing the MC packets out of order.

Latch-based TCAM and built-in self-test: The ternary content addressable memory is the key component in multicast routeing. It is designed without read out circuitry to save area. In SpiNNaker1, testing the TCAM involved a lot of human effort to design the test program. Therefore, a built-in self-test unit is devised to facilitate test automation and improve test efficiency.

8.5 The Processing Element (PE)

The key component of SpiNNaker2 is its processing element PE. The PE architecture is shown in Figure 8.3.

The PE is based on an ARM Cortex-M4 core with FPU. It contains 128 kBytes of local SRAM which is accessible as data and instruction memory by the processor. A crossbar handles local SRAM access from the processor core, the communication controller and from neighbouring PEs inside one QPE. Various components are

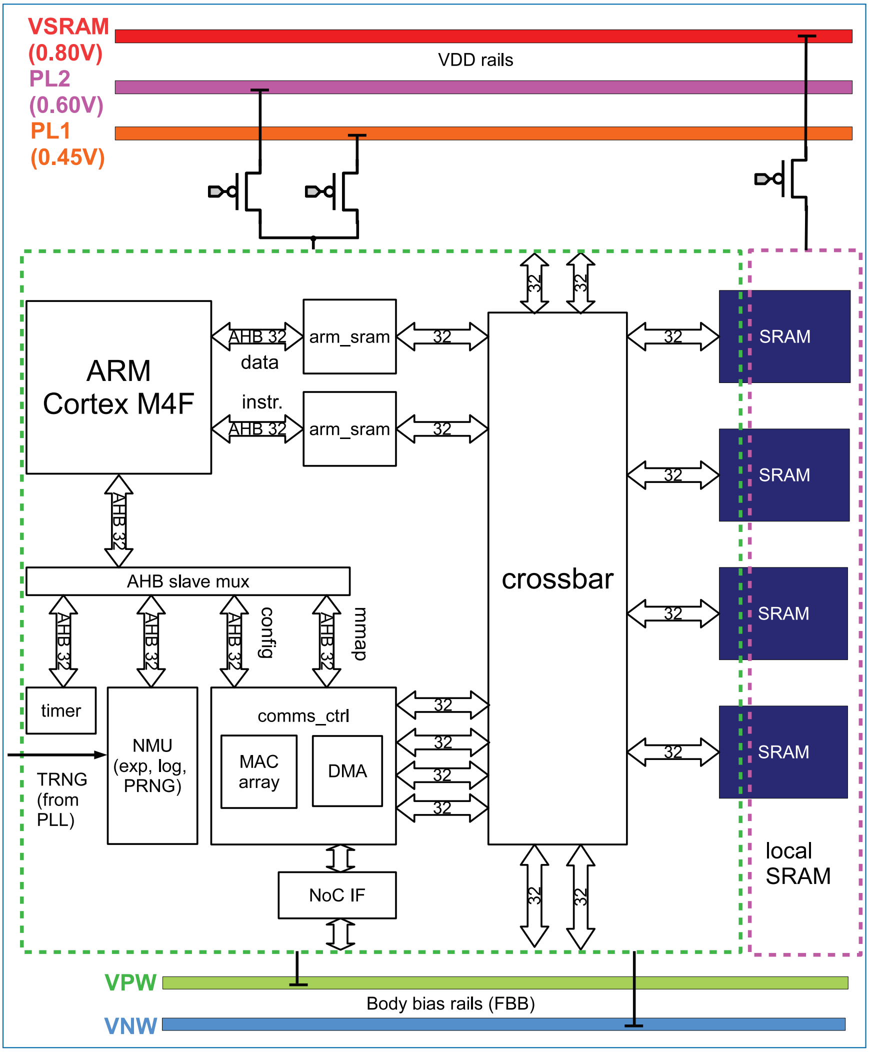


Figure 8.3. SpiNNaker2 PE architecture.

added to the PE to enhance its neuromorphic computation capabilities, as described below.

8.5.1 PE Components

Communications Controller

The Communications Controller is the interface between the PE and the NoC router. It is responsible for transmitting and receiving NoC packets to and from the communication network. It incorporates a bridge unit, a communication unit, a DMA engine, an ML accelerator, a response packet generator and an exception unit.

The bridge is the interface between the system bus of the ARM M4F and the NoC. It provides a method for the M4F to access remote registers or SRAM/SDRAM. A large part of the M4F system bus address space is bridged onto the NoC. The mechanism is implemented by suspending the M4F bus operation whilst a request packet is transmitted and a response packet is returned. Operation is therefore transparent to software apart from the speed of the operation. The corresponding hardware response packet generator unit in each PE translates request packets into local SRAM cycles and hence to response packets. As the data width of the system bus is 32 bits, the ARM M4F can only read or write a byte, half word or word through the bridge unit and the NoC. Note that there is an address translation from the local system bus address to the global NoC address.

The DMA unit is responsible for transferring data to/from remote SRAM/SDRAM memories. The transfer can be either reading, which transfers data from remote memories to local memories, or writing, which transfers data from local memories to remote memories. The transfer is at block level. Before the M4F initiates a transfer, some registers, such as the local start address, remote start address and the transfer length (in words), need to be configured.

The response packet generator answers requests from remote bridge or DMA units. This unit is not visible to the ARM M4F. Request packets arriving from the NoC are routed to this unit where they cause local bus activity – either read(s) or write(s). The appropriate response packets are generated and routed back to the source interface, as specified in the request. Zero (in the case of a write marked as 'buffered') up to eight (for a 128-byte read) response packets may be generated for each incoming request.

The communications unit within the SpiNNaker2 Communications Controller has three components: one default DMA (defaultDMA) and two SpiNNaker Packet DMAs (spDMAs). The two spDMAs are responsible for receiving specific types of SpiNNaker packet. There is a packet selector in each spDMA, which selects the type of packet that it accepts and rejects any other type of packet. If the packet is directed to the communications unit but neither spDMA0 nor spDMA1 accepts it, then it goes to the defaultDMA. The defaultDMA accepts any packets that the other components refuse to accept irrespective of packet type. In this way, no packet will be stuck at the NoC router, thus ensuring that the NoC always flows.

Random Number Accelerator

Spiking neural networks often make extensive use of random numbers, for example, to model intrinsic stochasticity or the influence of neighbouring brain areas that are not directly included in the model. While simple random number generators can be implemented cheaply in software, high-quality random sources are complex and much more efficient when implemented in hardware, so this is the solution adopted for SpiNNaker2.

The Pseudo-Random Number Generator (PRNG) chosen is a version of Marsaglia's KISS64 algorithm [155]. This has had several incarnations – we are using David Jones' implementation of what is called KISS99 here [133]. This is now the default PRNG on the SpiNNaker system, with a long cycle length $\approx 2^{123}$ and produces a high-quality random stream of 32-bit integers that satisfy the stringent TESTU01 BigCrush [133] and Dieharder [26] test suites.

In addition, a true random number is available, for exploring the impact of randomness on a neural network. The random source is provided by a *random bus*, which is driven from the noise generated in the phase-frequency detector of an All Digital Phase-Locked Loop (ADPLL) [177] used to generate the various clock signals required on the chip, and therefore incurs nearly zero overhead in terms of power and chip area.

The raw random data (after entropy extraction) is available directly and can also be used to scramble the seeds of the PRNGs from time to time. This is done by an XOR operation, which ensures an accumulation of entropy and is transparent to the user, which is why the migration from pseudo random to true random is a minor configuration change.

Rounding Accelerator

To improve the accuracy of a reduced precision neuron model, rounding can be done at scalar operation level [102]. To support this in the next-generation SpiNNaker chip, we are including a small hardware accelerator for stochastic rounding and round-to-nearest. Stochastic rounding in SpiNNaker is performed on fixed-point numbers by rounding them to a specified bit position (usually to fit a long number into 32 bits) probabilistically. The probability of rounding such a number up is proportional to the round-off residual, and to achieve this, a PRNG is used. Stochastically rounding fixed-point multiplication results has been shown to reduce numerical error in the Izhikevich neuron ODE solvers on SpiNNaker [102].

Elementary Function (exp, log) Accelerator

The exponential function is extensively used in neuroscience models, so an exponential accelerator will be added to the next-generation SpiNNaker chip. At the time of writing, prototype chips with such an accelerator have already been manufactured. The first prototype chip has a fixed-point version of the accelerator [188]. Using the accelerator on this chip, Yan *et al.* [266] demonstrated approximately 41% energy reduction in simulating a novel plasticity model. The second prototype has a fixed-point version with logarithm and accuracy control [164]. The final chip is currently planned to have both fixed- and floating-point exponential and logarithm functions with accuracy control in hardware and here we present some information about the design of this accelerator.

In the SpiNNaker1 system, there is no hardware support for transcendental functions, including exponentials, so the models that were developed used pre-computed Look-up tables (LUTs); this solution was explored in detail by Vogginger *et al.* [263]. The SpiNNaker compiler first takes a high-level description of the network dynamics specified by the user and pre-calculates a range of values of exponential decay for a specific time constant and a number of time instances on a fixed simulation time grid (either all possible times in a grid or a subset of times, depending on memory constraints). Then, the LUTs are copied into each core's local memory and used while the application is running.

However, this approach has two limitations:

- a limited number of timing constants and a limited input range can be used due to the constraints of the on-chip memory, and
- in the case where a model requires timing constants that depend on some dynamic quantity, such as the voltage-dependent timing constants in the intrinsic currents of the well-known *Hodgkin-Huxley* neuron model and its variants, the number of required look-up tables for each possible value that the time constant can take would be too large to store in the local SpiNNaker memory.

The memory requirements are further increased if the simulation time step is 0.1 ms, which is rarely used on SpiNNaker, but will be used on SpiNNaker2 as it will give more accuracy in all the parts of the simulation. In this scenario, the size of the LUTs for the same amount of time decay look-up will grow 10 times. For example, modelling a 16-bit exponential decay $e^{-\frac{\Delta t}{\tau_x}}$ for 1 second and all the values that Δt can take at 0.1 ms simulation time step will require 20 kB of memory space. A software exponential function is also available in the SpiNNaker software library, but with the latency of approximately 95 clock cycles it is a major limitation to real-time synaptic plasticity processing, where a single pair of spikes takes approximately 30 cycles (using LUTs for the exponential) as reported by Knight and Furber [127]. With most learning rules we usually require more than one exponential per spike pair processed. Learning rules requiring three or more decay time constants have already started appearing in the computational neuroscience literature and some have already been tried on SpiNNaker: see, for example, voltage-dependent STDP [37] implemented on SpiNNaker [69], the BCPNN learning rule [128, 263] and the neuromodulated STDP [165] learning rule.

Most of the algorithms for performing elementary functions are categorised into two types: *polynomial approximations* or *convergence algorithms* [56, 173]. For this accelerator, a well-known convergence algorithm [173] was chosen, which provides exponential and natural logarithm functions with overlapping hardware components. (Note that having both of these functions also allows us to derive a general power function for a limited range of arguments). The implementation is based on the iterative shift-add algorithms that are usually considered to be slower than polynomial approximation due to the serial dependencies in the algorithm, but they do not require multiplication, which reduces the area of the circuit. A further useful property of these iterative algorithms is that after just a few iterations they already contain an approximate result. This property is used to provide programmable accuracy control, following the principles of *approximate computing* [92] (in this case approximation comes not from the errors in the circuit as is most common, but by running fewer iterations than are required for a precise result) in order to add options for modellers to trade-off accuracy against speed and energy. This property will provide a platform for experimenting with concepts arising from the ongoing discussion about the maximum precision of arithmetic required in neuromorphic systems, for example, for representing weights in STDP [191] – the smaller the weight, the less precise the calculation of weight changes that is required.

Machine Learning (ML) Accelerator

In order to speed up machine learning algorithms, specifically DNNs, an accelerator for Multiply-Accumulate (MAc) operations is included in the PE. This accelerator performs matrix multiplication and convolution operations autonomously, off-loading significant work from the processor and thereby accelerating forward calculation of convolutional and fully-connected DNN layers. Moreover, matrix multiplication is a basic, calculation-intensive operation performed by many ML and data processing algorithms, and the accelerator can help speeding up these algorithms as well.

The structure of the ML accelerator is shown in Figure 8.4. It consists of a 16×4 array of MAc units operated by a data fetch and execution controller. Each MAc unit receives two 8-bit inputs and hosts a 29-bit accumulation register. Input data can be fetched from local SRAM and from the NoC, both connected via 128-bit wide interfaces. The resulting data are written to local SRAM.

The ML accelerator is configured and started either from the ARM processor via the AHB bus, or via the NoC. For matrix multiplication, memory addresses for the two input matrices and the output matrix, as well as dimensions of the matrices, have to be configured. Upon start, the ML accelerator fetches the next 128 bits of input data for each input matrix to two local registers and activates the MAc array. The processing follows an output-stationary data flow, that is, all MAc

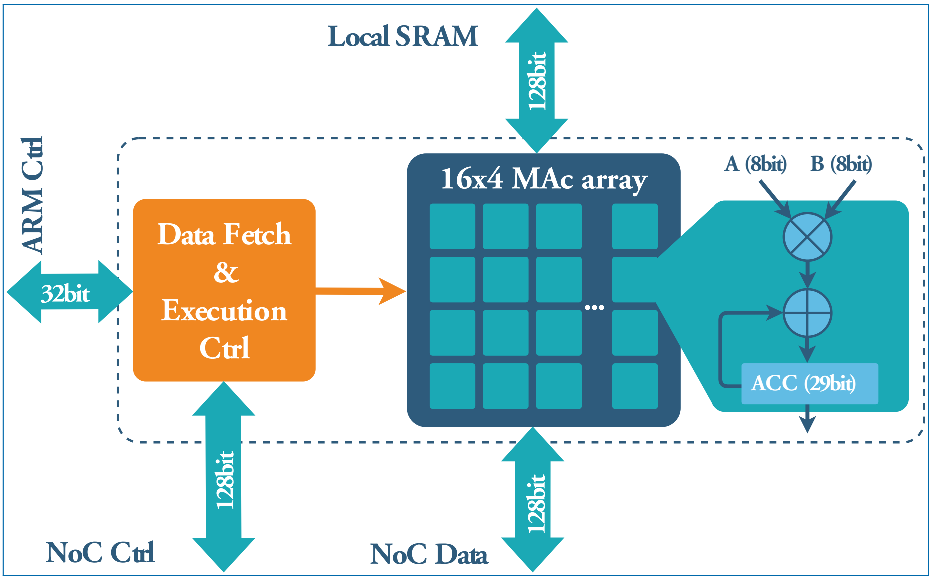


Figure 8.4. SpiNNaker2 machine learning accelerator.

operations for one output value are performed in one sequence and the output is written back afterwards. Due to the 16×4 arrangement of the MAc array, 128-bit data for the first operand and 32-bit data for the second operand are required in each clock cycle. Thus, the first operand fully occupies the bandwidth of either SRAM or NoC interface. If both operands are located in local SRAM or both are accessed via the NoC, MAc operations are interrupted periodically to fetch data for the second operand.

For a convolution operation, the dimensions of the convolution have to be configured besides the memory locations of input feature maps, kernels and output feature maps. Each row of the MAc array is used for a different output channel, that is, four output channels are calculated in parallel. Data flow is again output-stationary.

The ML accelerator supports both signed and unsigned inputs/output, which can be configured independently. Furthermore, input data width can be changed to 16-bit for either operand. This is realised with low overhead by adding together several MAc register results upon write-back. Also, configurable truncation and ReLU calculation are included in the write-back path. Output bit width can be chosen to be 8-, 16- or 32-bit, so that output data can be either directly used for the next DNN layer or further processed by the ARM core.

As a result, the wide-spread convolutional and fully-connected DNN layers with ReLU activation function can be processed completely by the machine learning accelerator, with the ARM core only configuring and starting it. Full flexibility for other layer types or activation functions (e.g. pooling layers or sigmoid activation functions) is still provided by the ARM core.

8.5.2 PE Implementation Strategy and Power Management

To achieve maximum energy efficiency, the PE is implemented for operation under ultra-low voltage (ULV) conditions with down to 0.50 V nominal supply voltage. This is enabled by the body bias capability of the 22-nm fully-depleted silicon-on-insulator (FDSOI) Complementary Metal Oxide Semiconductor (CMOS) technology [34], in combination with the Adaptive Body Biasing (ABB) technique [104]. As shown in Figure 8.3, the PE core logic standard cells in the ULV domain are connected to the body bias voltages VNW and VPW, which are adaptively controlled during system operation to compensate for Process, Voltage and Temperature (PVT) variations. The SRAMs are realised as dual rail macros where bitcells are operated from a 0.80 V nominal supply without body-biasing, and the SRAM periphery circuits are operated from the ULV domain with body bias. Therefore, robust ULV operation can be achieved, since the adaptation of the body bias voltages prevents near- or even sub-threshold operation, even at worst-case speed conditions (slow silicon, low supply voltage, minimum temperature). Following the approach from [104], the ABB technique is fully visible during physical implementation (logic synthesis, place and route, static timing analysis, power analysis), since the logic standard cells and SRAMs are characterised with the corresponding body bias values to the PVT corners. Therefore, synthesis and place and route can take advantage of the tightened PVT corners by means of ABB, which results in significantly improved power performance and area results. This is caused by the fact that the implementation tools can use logic cells with higher threshold voltage or longer gate length to meet the timing specification in the critical paths under worst-case speed conditions (slow silicon, low supply voltage, minimum temperature), thereby significantly reducing the leakage power consumption under worst-case power conditions (fast silicon, high supply voltage, maximum temperature).

Figure 8.5 shows the layout of the PE in 22FDX technology [34]. Iterative layout implementations with power analysis on the gate-level netlist have been executed for design space exploration. The main parameters are the nominal target supply voltage and the maximum constrained clock frequency. As an example, Figure 8.6 shows the simulation results for the energy per operation (normalised to the 0.60 V 300 MHz point) for a neuron state update kernel running on the PE. At low supply voltages down to 0.40 V, the maximum achievable clock frequency with reasonable cell leakage power drops significantly, resulting in more leakage energy accumulated per operation cycle, thereby increasing the energy-per-operation figure of merit. At

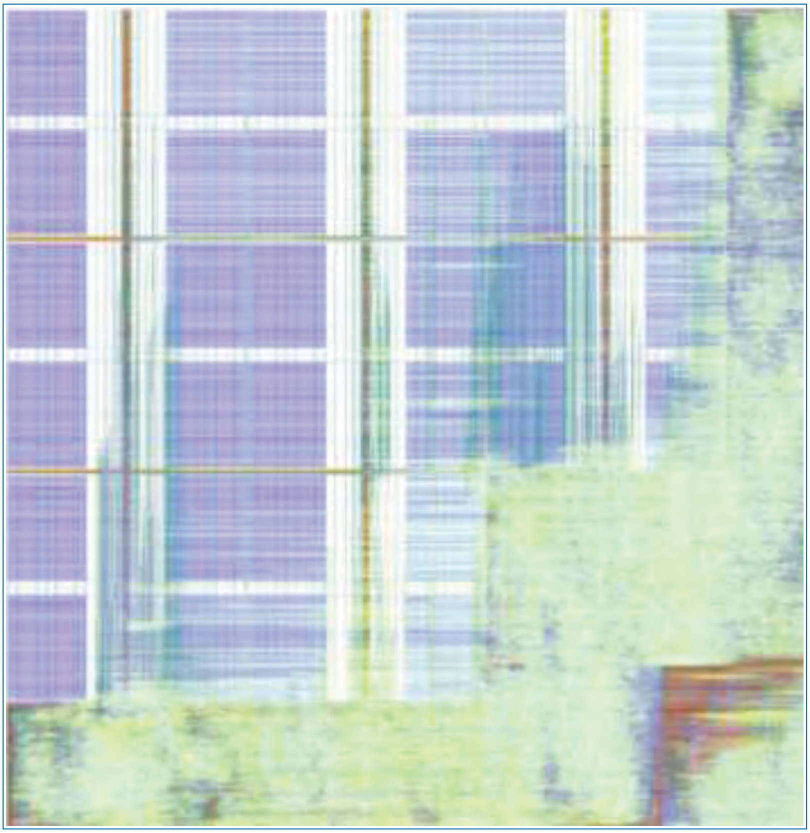


Figure 8.5. SpiNNaker2 PE layout in 22FDX technology.

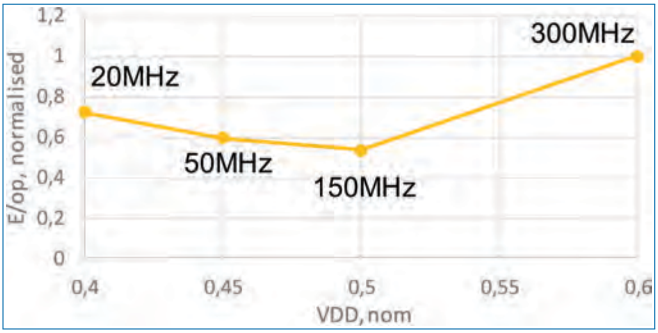


Figure 8.6. SpiNNaker2 PE relative energy per operation.

higher supply voltages, the dynamic power consumption dominates the total energy consumed. There exists a minimum energy point (MEP) around 0.50 V, where the PE implementation is capable of operating at 150 MHz. Note that at 0.50 V all standard cells are operating in a super-threshold regime for all PVT conditions, since the ABB approach adaptively compensates the device threshold voltages for PVT variations.

Although it is desired to operate the PE at the MEP for maximum efficiency, this obviously does not result in significant processor performance scaling compared to SpiNNaker1. Performance enhancement is achieved by applying Dynamic Voltage and Frequency Scaling (DVFS) [106, 107] to the PE. As shown in Figure 8.3 the PE core logic can be connected to one of two supply voltage rails. This allows for energy-efficient operation at a low-performance level at 0.50 V and peak performance operation at a higher-performance level at 0.60 V. It has been shown [106, 107] that under the dynamics of spiking neuromorphic applications, where peak processing power is only required in few simulation cycles, this technique significantly reduces the PE power consumption while still maintaining the temporal peak performance of the PE. The performance level transition is scheduled from a local power management controller at QPE level, based on the concept from [105]. Clocks are generated by PLLs [216]. Using this approach, each PE is capable of managing its own DVFS level just by knowing its local workload in the current simulation cycle (e.g. the number of synaptic events to be processed) independently of the other PEs. Performance level switching is realised in less than 100 ns, which is a negligible timing overhead compared to the neuromorphic real-time simulation with 0.1 ms or 1 ms timing resolution.

8.6 Summary

SpiNNaker2 represents the next step in the SpiNNaker story and brings us up to date. The SpiNNaker2 chip has yet (at the time of writing) to be fabricated, but will appear in 2020 and will form a new basis for the project for the future.

We have learnt a great deal in the 20 years that have gone into the project so far, about neuromorphic computing of course, but also about building large machines and making them reliable, about building large and complex software stacks, and about the areas of research of our users and collaborators in neuroscience and robotics. We have tried to capture those lessons, warts and all, in the accounts given in this book by the many contributing writers.

The book ends here, but the story goes on – there is still a great deal more to be learnt!

References

[1] Adams, S. V., A. D. Rast, C. Patterson, F. Galluppi, K. Brohan, J.-A. Pérez-Carrasco, T. Wennekers, S. Furber, and A. Cangelosi. 2014. "Towards Real-World Neurorobotics: Integrated Neuromorphic Visual Attention". In: *Neural Information Processing*. Ed. by C. K. Loo, K. S. Yap, K. W. Wong, A. T. Beng Jin, and K. Huang. Cham: Springer International Publishing. 563–570. ISBN: 978-3-319-12643-2.

[2] Alba, E. and J. M. Troya. 2001. "Analyzing synchronous and asynchronous parallel distributed genetic algorithms". *Future Generation Computer Systems*. Workshop on Bio-inspired Solutions to Parallel Computing problems 17(4): 451–465. ISSN: 0167-739X. DOI: 10.1016/S0167-739X(99)00129-6.

[3] Albada, S. J. V., A. G. Rowley, J. Senk, M. Hopkins, M. Schmidt, A. B. Stokes, D. R. Lester, M. Diesmann, and S. B. Furber. 2018. "Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model". *Frontiers in Neuroscience*. 12(5): 1–20. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00291.

[4] Amunts, K., C. Ebell, J. Muller, M. Telefont, A. Knoll, and T. Lippert. 2016. "The Human Brain Project: Creating a European research infrastructure to decode the human brain". *Neuron*. 92(3): 574–581. ISSN: 0896-6273. DOI: 10.1016/j.neuron.2016.10.046.

[5] Appel, K. I. and W. Haken. 1989. *Every Planar Map is Four Colorable*. Vol. 98. American Mathematical Soc.

[6] ARM. 2004. "ARM968E-S Technical Reference Manual". *ARM*. URL: https://static.docs.arm.com/ddi0311/d/DDI0311.pdf.

[7] Azevedo, F. A., L. R. Carvalho, L. T. Grinberg, J. M. Farfel, R. E. Ferretti, R. E. Leite, W. J. Filho, R. Lent, and S. Herculano-Houzel. 2009. "Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain". *Journal of Comparative Neurology*. 513(5): 532–541. ISSN: 00219967. DOI: 10.1002/cne.21974.

[8] Bamford, S. A., A. F. Murray, and D. J. Willshaw. 2010. "Synaptic rewiring for topographic mapping and receptive field development". *Neural Networks*. 23(4): 517–527. ISSN: 08936080. DOI: 10.1016/j. neunet.2010.01.005.

[9] Barahona, F. 1982. "On the computational complexity of Ising spin glass models". *Journal of Physics A: Mathematical and General*. 15(10): 3241.

[10] Bassett, D. S., D. L. Greenfield, A. Meyer-Lindenberg, D. R. Weinberger, S. W. Moore, and E. T. Bullmore. 2010. "Efficient physical embedding of topologically complex information processing networks in brains and computer circuits". *PLOS Computational Biology*. 6(4): 1–14. DOI: 10.1371/journal.pcbi.1000748.

[11] Becker, S. 2005. "A computational principle for hippocampal learning and neurogenesis". *Hippocampus*. 15(6): 722–738. ISSN: 10509631. DOI: 10.1002/hipo.20095. arXiv: 1507.07580.

[12] Bellec, G., D. Salaj, A. Subramoney, R. Legenstein, and W. Maass. 2018. "Long short-term memory and learning-to-learn in networks of spiking neurons". *arXiv:1803.09574 [cs, q-bio]*. arXiv: 1803.09574. URL: http://arxiv.org/abs/1803.09574.

[13] Bengio, Y., P. Lamblin, D. Popovici, and H. Larochelle. 2007. "Greedy layer-wise training of deep networks". In: *Advances in Neural Information Processing Systems*. 153–160.

[14] Bengio, Y., D. Lee, J. Bornschein, and Z. Lin. 2015. "Towards Biologically Plausible Deep Learning". *CoRR*. abs/1502.04156. arXiv: 1502.04156. URL: http://arxiv.org/abs/1502.04156.

[15] Beyer, H.-G. and H.-P. Schwefel. 2002. "Evolution strategies – A comprehensive introduction". *Natural Computing*. 1(1): 3–52. ISSN: 1572-9796. DOI: 10.1023/A:1015059928466.

[16] Bi, G. Q. and M. M. Poo. 1998. "Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type." *The Journal of Neuroscience: The Official Journal of the Society for Neuroscience*. 18(24): 10464–10472. ISSN: 0270-6474. DOI: 10.1038/25665.

[17] Bi, Z. 2018. *Finite Element Analysis Applications*. Elsevier. DOI: 10.1016/c2016-0-00054-2.

[18] Bienenstock, E. L., L. N. Cooper, and P. W. Munro. 1982. “Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex”. *The Journal of Neuroscience*. 2(1): 32–48.

[19] Binzegger, T., R. J. Douglas, and K. A. C. Martin. 2004. “A quantitative map of the circuit of cat primary visual cortex”. *Journal of Neuroscience*. 24(39): 8441–8453. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.1400-04.2004. eprint: https://www.jneurosci.org/content/24/39/8441.full.pdf.

[20] Bliss, T. V. and T. Lømo. 1973. “Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path”. *The Journal of Physiology*. 232(2): 331–356.

[21] Bogdan, P. A. 2020. “Structural Plasticity on SpiNNaker”. *PhD Thesis*. University of Manchester. 204. DOI: 10.13140/RG.2.2.33591.06568.

[22] Bogdan, P. A., G. Pineda-Garcıéa, S. Davidson, R. James, M. Hopkins, and S. Furber. 2019. “Event-based computation: Unsupervised elementary motion decomposition”. In: *Proceedings of the 2019 Emerging Technology Conference*. Ed. by M. Bane and V. Holmes. Huddersfield: EMiT/University of Huddersfield/High End Compute Ltd/University of Manchester. 20–23.

[23] Bogdan, P. A., A. G. D. Rowley, O. Rhodes, and S. Furber. 2018. “Structural plasticity on the SpiNNaker many-core neuromorphic system”. *Frontiers in Neuroscience*. 12(12): 1–20. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00434.

[24] Borst, A. and T. Euler. 2011. “Seeing things in motion: Models, circuits, and mechanisms”. *Neuron*. 71(6): 974–994. ISSN: 08966273. DOI: 10.1016/j.neuron.2011.08.031.

[25] Brown, A. D., S. Furber, J. S. Reeve, J. D. Garside, K. J. Dugan, L. A. Plana, and S. Temple. 2015. “SpiNNaker – Programming Model”. In: *IEEE Transactions on Computers*. Vol. 64. *6*. 1769–1782. DOI: 10.1109/TC.2014.2329686.

[26] Brown, R. G., D. Eddelbuettel, and D. Bauer. 2013. “Dieharder: A random number test suite”. *Open Source Software Library, Under Development*, URL: http://www.phy.duke.edu/~rgb/General/dieharder.php. URL: http://webhome.phy.duke.edu/%5C~%7B%7Drgb/General/dieharder.php.

[27] Brunel, N. 2000. “Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons”. *Journal of Computational Neuroscience*. 8: 193–208. ISSN: 09252312. DOI: 10.1016/S0925-2312(00)00179-X.

[28] Burnett, D. 2016. *The Idiot Brain: A Neuroscientist Explains What Your Head Is Really Up To*. HarperCollins, Canada. 336. ISBN: 978144345 0089.

[29] Butz, M. and A. van Ooyen. 2013. “A simple rule for dendritic spine and axonal bouton formation can account for cortical reorganization after focal retinal lesions”. *PLoS Computational Biology*. 9(10): 39–43. ISSN: 1553734X. DOI: 10.1371/journal.pcbi.1003259.

[30] Buzsáki, G. and K. Mizuseki. 2014. “The log-dynamic brain: How skewed distributions affect network operations”. *Nature Reviews Neuroscience*. 15(4): 264–278. ISSN: 14710048. DOI: 10.1038/nrn3687. arXiv: NIHMS150003.

[31] Camuñas-Mesa, L. A., Y. L. Domıénguez-Cordero, A. Linares-Barranco, T. Serrano-Gotarredona, and B. Linares-Barranco. 2018. “A configurable event-driven convolutional node with rate saturation mechanism for modular ConvNet systems implementation”. *Frontiers in Neuroscience*. 12: 63. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00063.

[32] Cao, Y., Y. Chen, and D. Khosla. 2015. “Spiking deep convolutional neural networks for energy-efficient object recognition”. *International Journal of Computer Vision*. 113(1): 54–66.

[33] Carnevale, N. T. and M. L. Hines. 2006. *The NEURON Book*. Cambridge University Press, New York, NY, USA. 1–457. ISBN: 9780511541612. DOI: 10.1017/CBO9780511541612. arXiv:1011.1669v3.

[34] Carter, R., J. Mazurier, L. Pirro, J. Sachse, P. Baars, J. Faul, C. Grass, G. Grasshoff, P. Javorka, T. Kammler, A. Preusse, S. Nielsen, T. Heller, J. Schmidt, H. Niebojewski, P. Chou, E. Smith, E. Erben, C. Metze, C. Bao, Y. Andee, I. Aydin, S. Morvan, J. Bernard, E. Bourjot, T. Feudel, D. Harame, R. Nelluri, H. -. Thees, L. M-Meskamp, J. Kluth, R. Mulfinger, M. Rashed, R. Taylor, C. Weintraub, J. Hoentschel, M. Vinet, J. Schaeffer, and B. Rice. 2016. “22nm FDSOI technology for emerging mobile, Internet-of-Things, and RF applications”. In: *2016 IEEE International Electron Devices Meeting (IEDM)*. 2.2.1–2.2.4. DOI: 10.1109/IEDM.2016.7838029.

[35] Chaitin, G. J. 1982. “Register allocation & spilling via graph coloring”. *SIGPLAN Not.* 17(6): 98–101. ISSN: 0362-1340. DOI: 10.1145/872726.806984.

[36] Christian, B. and T. Griffiths. 2016. *Algorithms to Live By: The Computer Science of Human Decisions*. New York, NY, USA: Henry Holt and Co., Inc. ISBN: 9781627790369.

[37] Clopath, C., L. Büsing, E. Vasilaki, and W. Gerstner. 2010. “Connectivity reflects coding: a model of voltage-based STDP with homeostasis.” *Nature Neuroscience*. 13(3): 344–52. ISSN: 1546-1726. DOI: 10.1038/nn.2479.

[38] Colbourn, C. 1984. “The complexity of completing partial Latin squares”. English (US). *Discrete Applied Mathematics*. 8(1): 25–30. ISSN: 0166-218X. DOI: 10.1016/0166-218X(84)90075-1.

[39] Crair, M. C. and W. Bialek. 1990. "Non-Boltzmann dynamics in networks of spiking neurons". In: *Advances in Neural Information Processing Systems*. 109–116.

[40] Curtis, A. R., T. Carpenter, M. Elsheikh, A. López-Ortiz, and S. Keshav. 2012. "Rewire: An optimization-based framework for unstructured data center network design". English. In: *INFOCOM, 2012 Proceedings IEEE*. 1116–1124.

[41] Dailey, D. P. 1980. "Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete". *Discrete Math.* 30(3): 289–293. ISSN: 0012-365X. DOI: 10.1016/0012-365X(80)90236-8.

[42] Dally, W. J. and B. Towles. 2004. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers.

[43] Darwin, C. and G. Beer, eds. 2008. *On the Origin of Species*. Revised Edition. *Oxford World's Classics*. Oxford, New York: Oxford University Press. ISBN: 978-0-19-921922-3.

[44] Davison, A. P., D. Brüderle, J. Eppler, J. Kremkow, E. Muller, and D. Pecevski. 2008. "PyNN: a common interface for neuronal network simulators". *Frontiers in Neuroinformatics*. 2(January): 1–10. ISSN: 16625196. DOI: 10.3389/neuro.11.011.2008.

[45] Dayan, P. and L. Abbott. 2002. "Theoretical neuroscience: Computational and mathematical modeling of neural systems (computational neuroscience)". *Journal of Cognitive Neuroscience*. 60(3): 480. DOI: 10.1016/j.neuron.2008.10.019. arXiv: 0-262-04199-5 [arXiv:gr-qc].

[46] Deb, K. and T. Goel. 2001. "Controlled Elitist Non-dominated Sorting Genetic Algorithms for Better Convergence". In: *Evolutionary Multi-Criterion Optimization*. Ed. by E. Zitzler, L. Thiele, K. Deb, C. A. Coello Coello, and D. Corne. Berlin, Heidelberg: Springer Berlin Heidelberg. 67–81. ISBN: 978-3-540-44719-1.

[47] Delbruck, T. and P. Lichtsteiner. 2007. "Fast sensory motor control based on event-based hybrid neuromorphic-procedural system". In: *2007 IEEE International Symposium on Circuits and Systems*. 845–848. DOI: 10.1109/ISCAS.2007.378038.

[48] Denk, C., F. Llobet-Blandino, F. Galluppi, L. A. Plana, S. Furber, and J. Conradt. 2013. "Real-Time Interface Board for Closed-Loop Robotic Tasks on the SpiNNaker Neural Computing System". In: *Artificial Neural Networks and Machine Learning – ICANN 2013*. Ed. by V. Mladenov, P. Koprinkova-Hristova, G. Palm, A. E. P. Villa, B. Appollini, and N. Kasabov. Berlin, Heidelberg: Springer Berlin Heidelberg. 467–474. ISBN: 978-3-642-40728-4.

[49] Destexhe, A., Z. F. Mainen, and T. J. Sejnowski. 2002. “Kinetic models for synaptic interactions”. *The Handbook of Brain Theory and Neural Networks (2nd ed)*: 1126–1130.

[50] Diehl, P. U. and M. Cook. 2014. “Efficient implementation of STDP rules on SpiNNaker neuromorphic hardware”. In: *Proceedings of the International Joint Conference on Neural Networks*. Beijing, China. 4288–4295. ISBN: 9781479914845. DOI: 10.1109/IJCNN. 2014.6889876.

[51] Diehl, P. U., D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer. 2015. “Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing”. In: *Neural Networks (IJCNN), 2015 International Joint Conference on*. IEEE. 1–8.

[52] Dominguez-Morales, J. P., Q. Liu, R. James, D. Gutierrez-Galan, A. Jimenez-Fernandez, S. Davidson, and S. Furber. 2018. “Deep Spiking Neural Network model for time-variant signals classification: a real-time speech recognition approach”. In: *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 1–8.

[53] Eliasmith, C. 2013. *How to Build a Brain: A Neural Architecture for Biological Cognition*. Oxford University Press.

[54] Eliasmith, C., T. C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, and D. Rasmussen. 2012. “A large-scale model of the functioning brain”. *Science*. 338(6111): 1202–1205. DOI: 10.1126/science.1225266.

[55] Elliott, T. and N. R. Shadbolt. 1999. “A neurotrophic model of the development of the retinogeniculocortical pathway induced by spontaneous retinal waves.” *Journal of Neuroscience*. 19(18): 7951–7970. ISSN: 1529-2401. URL: https://www.jneurosci.org/content/19/18/7951.

[56] Ercegovac, M. and T. Lang. 2004. *Digital Arithmetic*. *Morgan Kaufmann Series in Comp*. Morgan Kaufmann. ISBN: 9781558607989. URL: https://books.google.co.uk/books?id=uUk%5C_AQAAIAAJ.

[57] Ercsey-Ravasz, M. and Z. Toroczkai. 2012. “The chaos within Sudoku”. *Scientific Reports*. 2: 725.

[58] Euler, T., S. Haverkamp, T. Schubert, and T. Baden. 2014. “Retinal bipolar cells: elementary building blocks of vision”. *Nature Reviews Neuroscience*. 15(8): 507–519.

[59] Fisher, S. D., P. B. Robertson, M. J. Black, P. Redgrave, M. A. Sagar, W. C. Abraham, and J. N. Reynolds. 2017. “Reinforcement determines the timing dependence of corticostriatal synaptic plasticity in vivo”. *Nature Communications*. 8(1). ISSN: 20411723. DOI: 10.1038/s41467-017-00394-x.

[60] Florian, R. V. 2005. “A reinforcement learning algorithm for spiking neural networks”. *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC’05)*: 8 pp. DOI: 10.1109/SYNASC.2005.13.

[61] Fonseca Guerra, G. A. and S. Furber. 2017. “Using stochastic spiking neural networks on SpiNNaker to solve constraint satisfaction problems”. *Frontiers in Neuroscience*. 11(12). ISSN: 1662453X. DOI: 10.3389/fnins.2017.00714.

[62] Francis, H. 2001. “ARM DSP-Enhanced Extensions”. URL: https://pdfs.semanticscholar.org/5e30/761e1b43b3ad93270940bbe31a74d9c7fa20.pdf.

[63] Fukushima, K. and S. Miyake. 1982. “Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition”. In: *Competition and Cooperation in Neural Nets*. Springer. 267–285.

[64] Furber, S. B., F. Galluppi, S. Temple, and L. A. Plana. 2014. “The SpiNNaker project”. *Proceedings of the IEEE*. 102(5): 652–665. ISSN: 00189219. DOI: 10.1109/JPROC.2014.2304638. arXiv: 1011.1669v3.

[65] Furber, S. B., D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown. 2013. “Overview of the SpiNNaker system architecture”. *IEEE Transactions on Computers*. 62(12): 2454–2467. ISSN: 00189340. DOI: 10.1109/TC.2012.142.

[66] Furukawa, H., S. K. Singh, R. Mancusso, and E. Gouaux. 2005. “Subunit arrangement and function in NMDA receptors”. *Nature*. 438(7065): 185.

[67] Galluppi, F., K. Brohan, S. Davidson, T. Serrano-Gotarredona, J.-A. P. Carrasco, B. Linares-Barranco, and S. Furber. 2012. “A real-time, event-driven neuromorphic system for goal-directed attentional selection”. In: *Neural Information Processing*. Springer. 226–233.

[68] Galluppi, F., S. Davies, A. Rast, T. Sharp, L. A. Plana, and S. Furber. 2012. “A hierarchical configuration system for a massively parallel neural hardware platform”. *Proceedings of the 9th conference on Computing Frontiers – CF ’12*: 183. DOI: 10.1145/2212908.2212934.

[69] Galluppi, F., X. Lagorce, E. Stromatias, M. Pfeiffer, L. A. Plana, S. Furber, and R. B. Benosman. 2015. “A framework for plasticity implementation on the SpiNNaker neural architecture”. *Frontiers in Neuroscience*. 9(JAN): 1–20. ISSN: 1662453X. DOI: 10.3389/fnins.2014.00429.

[70] Gardner, B. and A. Grüning. 2013. “Learning Temporally Precise Spiking Patterns through Reward Modulated Spike-Timing-Dependent Plasticity”. In: *Artificial Neural Networks and Machine Learning – ICANN 2013*. Ed. by V. Mladenov, P. Koprinkova-Hristova, G. Palm, A. E. P. Villa, B. Appollini, and N. Kasabov. Berlin, Heidelberg: Springer Berlin Heidelberg. 256–263. ISBN: 978-3-642-40728-4.

[71] Gardner, M. 1970. "Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life"". *Scientific American*. 223: 120–123. URL: http://www.worldcat.org/isbn/0894540017.

[72] Garside, J. D., S. B. Furber, S. Temple, D. M. Clark, and L. A. Plana. 2012. "An asynchronous fully digital delay locked loop for DDR SDRAM data recovery". In: *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*. 49–56. DOI: 10.1109/ASYNC.2012.18.

[73] George, R., G. Indiveri, and S. Vassanelli. 2017. "Activity dependent structural plasticity in neuromorphic systems". In: *Biomedical Circuits and Systems Conference (BioCAS)*. Torino, Italy: IEEE. 1–4. DOI: 10.1109/BIOCAS.2017.8325074.

[74] Gerstner, W., W. M. Kistler, R. Naud, and L. Paninski. 2014. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press. 1–577. ISBN: 9781107447615. DOI: 10.1017/CBO9781107447615.

[75] Gerstner, W., M. Lehmann, V. Liakoni, D. Corneil, and J. Brea. 2018. "Eligibility traces and plasticity on behavioral time scales: Experimental support of neoHebbian three-factor learning rules". *Frontiers in Neural Circuits*. 12(7): 1–16. ISSN: 1662-5110. DOI: 10.3389/fncir.2018.00053. arXiv: 1801.05219.

[76] Gewaltig, M.-O. and M. Diesmann. 2007. "NEST (NEural Simulation Tool)". *Scholarpedia*. 2(4): 1430. ISSN: 1941-6016. DOI: 10.4249/scholarpedia.1430.

[77] Gewaltig, M.-O. and M. Diesmann. 2007. "NEST (NEural Simulation Tool)". *Scholarpedia*. 2(4): 1430.

[78] Gibbs, M. E., D. Hutchinson, and L. Hertz. 2008. "Astrocytic involvement in learning and memory consolidation". *Neuroscience and Biobehavioral Reviews*. 32(5): 927–944. ISSN: 0149-7634. DOI: 10.1016/j.neubiorev.2008.02.001.

[79] Gittis, A. H., A. B. Nelson, M. T. Thwin, J. J. Palop, and A. C. Kreitzer. 2010. "Distinct roles of GABAergic interneurons in the regulation of striatal output pathways". *Journal of Neuroscience*. 30(6): 2223–2234. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.4870-09.2010.

[80] Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. 2014. "Generative adversarial nets". In: *Advances in Neural Information Processing Systems*. 2672–2680.

[81] Goodman, D. F. M. and R. Brette. 2008. "Brian: a simulator for spiking neural networks in Python". *Frontiers in Neuroinformatics*. 2(5). ISSN: 1662-5196. DOI: 10.3389/neuro.11.005.2008.

[82] Goodman, D. F. and R. Brette. 2009. "The brian simulator". *Frontiers in Neuroscience*. 3(9): 192–197. ISSN: 1662-453X. DOI: 10.3389/neuro.01.026.2009.

[83] Graves, A. and N. Jaitly. 2014. "Towards end-to-end speech recognition with recurrent neural networks." In: *ICML*. Vol. 14. 1764–1772.

[84] Gurney, K., T. J. Prescott, and P. Redgrave. 2001. "A computational model of action selection in the basal ganglia. I. A new functional anatomy." *Biological Cybernetics*. 84(6): 401–410. ISSN: 0340-1200. DOI: 10.1007/PL00007984.

[85] Gurney, K., T. J. Prescott, and P. Redgrave. 2001. "A computational model of action selection in the basal ganglia. II. Analysis and simulation of behaviour." *Biological Cybernetics*. 84(6): 411–423. ISSN: 0340-1200. DOI: 10.1007/PL00007985.

[86] Gütig, R., R. Aharonov, S. Rotter, H. Sompolinsky, R. Gu, R. Aharonov, S. Rotter, and H. Sompolinsky. 2003. "Learning input correlations through nonlinear temporally asymmetric Hebbian plasticity". *The Journal of Neuroscience: The Official Journal of the Society for Neuroscience*. 23(9): 3697–3714. ISSN: 1529-2401. URL: http://www.ncbi.nlm.nih.gov/pubmed/12736341.

[87] Haas, S., O. Arnold, B. Nöthen, S. Scholze, G. Ellguth, A. Dixius, S. Höppner, S. Schiefer, S. Hartmann, S. Henker, T. Hocker, J. Schreiter, H. Eisenreich, J. U. Schlüßler, D. Walter, T. Seifert, F. Pauls, M. Hasler, Y. Chen, H. Hensel, S. Moriam, E. Matús, C. Mayr, R. Schüffny, and G. P. Fettweis. 2016. "An MPSoC for energy-efficient database query processing". In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. DOI: 10.1145/2897937.2897986.

[88] Haas, S., T. Seifert, B. Nöthen, S. Scholze, S. Höppner, A. Dixius, E. P. Adeva, T. Augustin, F. Pauls, S. Moriam, M. Hasler, E. Fischer, Y. Chen, E. Matúš, G. Ellguth, S. Hartmann, S. Schiefer, L. Cederström, D. Walter, S. Henker, S. Hänzsche, J. Uhlig, H. Eisenreich, S. Weithoffer, N. Wehn, R. Schüffny, C. Mayr, and G. Fettweis. 2017. "A heterogeneous SDR MPSoC in 28 nm CMOS for low-latency wireless applications". In: *Proceedings of the 54th Annual Design Automation Conference 2017. DAC '17*. Austin, TX, USA: ACM. 47:1–47:6. ISBN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3062188.

[89] Habenschuss, S., Z. Jonke, and W. Maass. 2013. "Stochastic computations in cortical microcircuit models". *PLOS Computational Biology*. 9(11): 1–28. DOI: 10.1371/journal.pcbi.1003311.

[90] Hagmann, P., L. Cammoun, X. Gigandet, R. Meuli, C. J. Honey, V. J. Wedeen, and O. Sporns. 2008. “Mapping the structural core of human cerebral cortex”. *PLOS Biology*. 6(7): 1–15. DOI: 10.1371/journal.pbio.0060159.

[91] Halassa, M. M., T. Fellin, and P. G. Haydon. 2007. “The tripartite synapse: roles for gliotransmission in health and disease”. *Trends in Molecular Medicine*. 13(2): 54–63. ISSN: 1471-4914. DOI: 10.1016/j.molmed.2006.12.005.

[92] Han, J. and M. Orshansky. 2013. “Approximate computing: An emerging paradigm for energy-efficient design”. In: *2013 18th IEEE European Test Symposium (ETS)*. 1–6. DOI: 10.1109/ETS.2013.6569370.

[93] He, K., X. Zhang, S. Ren, and J. Sun. 2016. “Deep residual learning for image recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. DOI: 10. 1109/CVPR.2016.90.

[94] Heathcote, J. 2016. “Building and operating large-scale SpiNNaker machines”. *PhD Thesis*. University of Manchester, UK. URL: http://www.manchester.ac.uk/escholar/uk-ac-man-scw:305482.

[95] Heathcote, J. 2016. “Spalloc: SpiNNaker machine partitioning and allocation server”, Available online. URL: https://github.com/SpiNNakerManchester/spalloc_server.

[96] Heathcote, J. 2016. “SpiNNer: SpiNNaker wiring tool”, Available online. URL: https://github.com/SpiNNakerManchester/SpiNNer.

[97] Heitger, F., L. Rosenthaler, R. V. D. Heydt, E. Peterhans, and O. Kübler. 1992. “Simulation of neural contour mechanisms: from simple to end-stopped cells”. *Vision Research*. 32(5): 963–981. ISSN: 0042-6989. DOI: https://doi.org/10.1016/0042-6989(92)90039-L.

[98] Hinton, G., S. Osindero, and Y. W. Teh. 2006. “A fast learning algorithm for deep belief nets.” *Neural Computation*. 18(7): 1527–1554.

[99] Hochreiter, S. and J. Schmidhuber. 1997. “Long short-term memory”. *Neural Computation*. 9(8): 1735–1780.

[100] Hodgkin, A. L. and A. F. Huxley. 1952. “A quantitative description of membrane current and its application to conduction and excitation in nerve”. *The Journal of Physiology*. 117(4): 500.

[101] Hopkins, M. and S. Furber. 2015. “Accuracy and efficiency in fixedpoint neural ODE solvers”. *Neural Computation*. 27(10): 2148–2182.

[102] Hopkins, M., M. Mikaitis, D. R. Lester, and S. Furber. 2020. “Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations”. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*. 378(2166): 20190052. DOI: 10.1098/rsta.2019.0052.

[103] Hopkins, M., G. Pineda-Garcıéa, P. A. Bogdan, and S. Furber. 2018. "Spiking neural networks for computer vision". *Interface Focus*. 8(4). ISSN: 2042-8898. DOI: 10.1098/rsfs.2018.0007.

[104] Höppner, S., H. Eisenreich, D. Walter, U. Steeb, A. Scharfe, C. Dmello, R. Sinkwitz, H. Bauer, A. Oefelein, F. Schraut, J. Schreiter, R. Niebsch, S. Scherzer, M. Orgis, U. Hensel, and J. Winkler. 2019. "How to achieve world-leading energy efficiency using 22FDX with adaptive body biasing on an arm Cortex-M4 IoT SoC". In: *ESSDERC Conference 2019 – 49st European Solid-State Device Research Conference (ESSDERC)*.

[105] Höppner, S., C. Shao, H. Eisenreich, G. Ellguth, M. Ander, and R. Schüffny. 2012. "A power management architecture for fast per-core DVFS in heterogeneous MPSoCs". In: *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*. 261–264. DOI: 10.1109/ISCAS.2012.6271840.

[106] Höppner, S., B. Vogginger, Y. Yan, A. Dixius, S. Scholze, J. Partzsch, F. Neumärker, S. Hartmann, S. Schiefer, G. Ellguth, L. Cederstroem, L. A. Plana, J. Garside, S. Furber, and C. Mayr. 2019. "Dynamic power management for neuromorphic many-core systems". *IEEE Transactions on Circuits and Systems I: Regular Papers*. 66(8): 2973–2986. DOI: 10.1109/TCSI.2019.2911898.

[107] Höppner, S., Y. Yan, B. Vogginger, A. Dixius, J. Partzsch, F. Neumärker, S. Hartmann, S. Schiefer, S. Scholze, G. Ellguth, L. Cederstroem, M. Eberlein, C. Mayr, S. Temple, L. A. Plana, J. Garside, S. Davison, D. R. Lester, and S. Furber. 2017. "Dynamic voltage and frequency scaling for neuromorphic many-core systems". In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4. DOI: 10.1109/ISCAS.2017.8050656.

[108] Howard, A. G., M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. 2017. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[109] Hubel, D. H. and T. N. Wiesel. 1962. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". *The Journal of Physiology*. 160(1): 106–154.

[110] Humphries, M. D., R. D. Stewart, and K. N. Gurney. 2006. "A physiologically plausible model of action selection and oscillatory activity in the basal ganglia". *Journal of Neuroscience*. 26(50): 12921–12942. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.3486-06.2006.

[111] Hunsberger, E. and C. Eliasmith. 2015. "Spiking deep networks with LIF neurons". *arXiv preprint*.

[112] Inayat, S., J. Barchini, H. Chen, L. Feng, X. Liu, J. Cang, X. S. Inayat, J. Barchini, X. H. Chen, L. Feng, X. Liu, and X. J. Cang. 2015. "Neurons in the most superficial lamina of the mouse superior colliculus are highly selective for stimulus direction". *Journal of Neuroscience*. 35(20): 7992–8003. ISSN: 02706474. DOI: 10.1523/JNEUROSCI.0173-15. 2015.

[113] Inc., W. R. 2018. "Mathematica, Version 11.3". Champaign, IL.

[114] Indiveri, G., B. Linares-Barranco, T. J. Hamilton, A. Van Schaik, R. Etienne-Cummings, T. Delbruck, S.-C. Liu, P. Dudek, P. Häfliger, S. Renaud, *et al.* 2011. "Neuromorphic silicon neuron circuits". *Frontiers in Neuroscience*. 5(OCT): 1–2. ISSN: 1662-4548. DOI: 10.3389/fnins.2011. 00118.

[115] Iyer, R., V. Menon, M. Buice, C. Koch, and S. Mihalas. 2013. "The influence of synaptic weight distribution on neuronal population dynamics". *PLOS Computational Biology*. 9(10): 1–16. DOI: 10.1371/journal.pcbi.1003248.

[116] Izhikevich, E. 2003. "Simple model of spiking neurons". *IEEE Transactions on Neural Networks*. 14(6): 1569–1572. ISSN: 1045-9227. DOI: 10.1109/TNN.2003.820440.

[117] Izhikevich, E. M. 2007. "Solving the distal reward problem through linkage of STDP and dopamine signaling". *BMC Neuroscience*. 8(2): S15. ISSN: 1471-2202. DOI: 10.1186/1471-2202-8-S2-S15.

[118] Izhikevich, E. M. 2007. "Solving the distal reward problem through linkage of STDP and dopamine signaling". *Cerebral Cortex*. 17(10): 2443–2452. ISSN: 10473211. DOI: 10.1093/cercor/bhl152.

[119] James, R., J. Garside, L. A. Plana, A. Rowley, and S. B. Furber. 2018. "Parallel distribution of an inner hair cell and auditory nerve model for real-time application". *IEEE Transactions on Biomedical Circuits and Systems*: 1–9. ISSN: 1932-4545. DOI: 10.1109/TBCAS. 2018.2847562.

[120] James, R., J. Garside, M. Hopkins, L. A. Plana, S. Temple, S. Davidson, and S. Furber. 2018. "Parallel distribution of an inner hair cell and auditory nerve model for real-time application". *2017 IEEE Biomedical Circuits and Systems Conference, BioCAS 2017 – Proceedings*. 2018-Janua(5): 1–4. ISSN: 19324545. DOI: 10.1109/BIOCAS.2017.8325171.

[121] Jin, X., A. Rast, F. Galluppi, S. Davies, and S. Furber. 2010. "Implementing spike-timing-dependent plasticity on SpiNNaker neuromorphic hardware". In: *Proceedings of the International Joint Conference on Neural Networks*. Barcelona, Spain: IEEE. 1–8. ISBN: 9781424469178. DOI: 10.1109/IJCNN.2010.5596372.

[122] Jug, F., M. Cook, and A. Steger. 2012. "Recurrent competitive networks can learn locally excitatory topologies". In: *Proceedings of 2012 International Joint Conference on Neural Networks (IJCNN)*. 1–8. DOI: 10.1109/IJCNN.2012.6252786.

[123] Kaas, J. H. 1997. "Topographic maps are fundamental to sensory processing". *Brain Research Bulletin*. 44(2): 107–112. ISSN: 03619230. DOI: 10.1016/S0361-9230(97)00094-4.

[124] Kappel, D., S. Habenschuss, R. Legenstein, and W. Maass. 2015. "Network plasticity as Bayesian inference". *PLoS Computational Biology*. 11(11): 1–31. ISSN: 15537358. DOI: 10.1371/journal.pcbi. 1004485. arXiv:1504.05143v1.

[125] Karpathy, A. and L. Fei-Fei. 2015. "Deep visual-semantic alignments for generating image descriptions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3128–3137.

[126] King, A. J. 2004. "The superior colliculus." *Current Biology: CB*. 14(9): R335–R338. ISSN: 0960-9822. DOI: 10.1016/j.cub.2004.04.018.

[127] Knight, J. C. and S. Furber. 2016. "Synapse-centric mapping of cortical models to the SpiNNaker neuromorphic architecture". *Frontiers in Neuroscience*. 10(9): 1–14. ISSN: 1662453X. DOI: 10.3389/fnins.2016.00420.

[128] Knight, J. C., P. J. Tully, B. A. Kaplan, A. Lansner, and S. B. Furber. 2016. "Large-scale simulations of plastic neural networks on neuromorphic hardware." *Frontiers in Neuroanatomy*. 10(April): 37. ISSN: 1662-5129. DOI: 10.3389/fnana.2016.00037.

[129] Knight, J. C. 2016. "Plasticity in large-scale neuromorphic models of the neocortex". *PhD Thesis*. University of Manchester. 169.

[130] Knuth, D. E. 1969. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. Addison-Wesley.

[131] Kohonen, T. 1982. "Self-organized formation of topologically correct feature maps". *Biological Cybernetics*. 43(1): 59–69. ISSN: 0340-1200. DOI: 10.1007/BF00337288.

[132] Krizhevsky, A., I. Sutskever, and G. E. Hinton. 2012. "ImageNet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems*. 1097–1105.

[133] L'Ecuyer, P. and R. Simard. 2007. "TestU01: A C Library for Empirical Testing of Random Number Generators". *ACM Transactions on Mathematical Software*. 33(4): 22:1–22:40. ISSN: 0098-3500. DOI: 10.1145/1268776.1268777.

[134] La Camera, G., M. Giugliano, W. Senn, and S. Fusi. 2008. "The response of cortical neurons to in vivo-like input current: theory and experiment". *Biological Cybernetics*. 99(4–5): 279–301.

[135] Lanzerotti, M. Y., G. Fiorenza, and R. A. Rand. 2005. "Microminiature packaging and integrated circuitry: The work of E. F. Rent, with an application to on-chip interconnection requirements". *IBM Journal of Research and Development*. 49(4.5): 777–803. ISSN: 0018-8646. DOI: 10.1147/rd.494.0777.

[136] Lazzaro, J., J. Wawrzynek, M. Mahowald, M. Sivilotti, and D. Gillespie. 1993. "Silicon auditory processors as computer peripherals". *IEEE Transactions on Neural Networks*. 4(3): 523–528. ISSN: 1045-9227. DOI: 10.1109/72.217193.

[137] Le, Q. V. 2013. "Building high-level features using large scale unsupervised learning". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE. 8595–8598.

[138] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. "Gradient-based learning applied to document recognition". *Proceedings of the IEEE*. 86(11): 2278–2324. ISSN: 00189219. DOI: 10.1109/5.726791.

[139] LeCun, Y. and Y. Bengio. 1995. "Convolutional networks for images, speech, and time series". *The Handbook of Brain Theory and Neural Networks*. 3361(10): 1995.

[140] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. "Gradient-based learning applied to document recognition". *Proceedings of the IEEE*. 86(11): 2278–2324.

[141] Lenero-Bardallo, J. A., T. Serrano-Gotarredona, and B. Linares-Barranco. 2011. "A 3.6μs latency asynchronous frame-free event-driven dynamic-vision-sensor". *IEEE Journal of Solid-State Circuits*. 46(6): 1443–1455. ISSN: 0018-9200. DOI: 10.1109/JSSC.2011.2118490.

[142] Levy, W. and O. Steward. 1983. "Temporal contiguity requirements for long-term associative potentiation/depression in the hippocampus". *Neuroscience*. 8(4). URL: http://www.sciencedirect.com/science/article/pii/0306452283900106.

[143] Lichtsteiner, P., C. Posch, and T. Delbruck. 2006. "A 128 × 128 120 db 30 mw asynchronous vision sensor that responds to relative intensity change". In: *Solid-State Circuits Conference, 2006. ISSCC 2006. Digest of Technical Papers. IEEE International*. IEEE. 2060–2069.

[144] Liu, G., P. Camilleri, S. Furber, and J. Garside. 2015. "Network traffic exploration on a many-core computing platform: SpiNNaker real-time traffic visualiser". English. In: *11th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*. DOI: 10.1109/PRIME.2015.7251376.

[145] Liu, G., J. Garside, S. Furber, L. Plana, and D. Koch. 2017. "Asynchronous interface FIFO design on FPGA for high-throughput NRZ synchronisation". English. In: *The 27th International Conference on Field Programmable Logic and Applications*. DOI: 10.23919/FPL.2017.8056801.

[146] Liu, Q. 2018. "Deep spiking neural networks". *PhD Thesis*. University of Manchester. 212.

[147] Liu, Q. and S. Furber. 2016. "Noisy Softplus: a biology inspired activation function". In: *International Conference on Neural Information Processing*. Springer. 405–412.

[148] Liu, Q., G. Pineda-Garcıéa, E. Stromatias, T. Serrano-Gotarredona, and S. B. Furber. 2016. "Benchmarking spike-based visual recognition: a dataset and evaluation". *Frontiers in Neuroscience*. 10.

[149] Liu, Y. H. and X. J. Wang. 2001. "Spike-frequency adaptation of a generalized leaky integrate-and-fire model neuron". *Journal of Computational Neuroscience*. 10(1): 25–45. ISSN: 09295313. DOI: 10.1023/A: 1008916026143.

[150] Lopez-Poveda, E. A. and R. Meddis. 2001. "A human nonlinear cochlear filterbank". *The Journal of the Acoustical Society of America*. 110(6): 3107–3118.

[151] Lowe, D. G. 1999. "Object recognition from local scale-invariant features". In: *The Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999*. Vol. 2. IEEE. 1150–1157.

[152] Mahowald, M. 1992. "VLSI analogs of neuronal visual processing: A synthesis of form and function". *Technology*. 1992(5): 236. URL: http://caltechcstr.library.caltech.edu/591/.

[153] Malmierca, M. S., L. A. Anderson, and F. M. Antunes. 2015. "The cortical modulation of stimulus-specific adaptation in the auditory midbrain and thalamus: a potential neuronal correlate for predictive coding". *Frontiers in Systems Neuroscience*. 9.

[154] Markram, H., J. Lubke, M. Frotscher, and B. Sakmann. 1997. "Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs". *Science*. 275(5297): 213–215. ISSN: 00368075. DOI: 10.1126/science.275.5297.213.

[155] Marsaglia, G. and A. Zaman. 1993. "The KISS generator". *Tech. rep.*, Department of Statistics, University of Florida.

[156] Mazaris, D. 1997. "The reality of patch-cord management". *Cabling Installation & Maintenance*. Feb.

[157] Mazurek, M., M. Kager, and S. D. V. Hooser. 2014. "Robust quantification of orientation selectivity and direction selectivity". *Frontiers in Neural Circuits*. DOI: 10.3389/fncir.2014.00092.

[158] Mead, C. 1989. *Analog VLSI and Neural Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-05992-4.

[159] Meddis, R., W. Lecluyse, N. R. Clark, T. Jürgens, C. M. Tan, M. R. Panda, and G. J. Brown. 2013. "A computer model of the auditory periphery and its application to the study of hearing". In: *Basic Aspects of Hearing*. Springer. 11–20.

[160] Mel, B. W. 1992. “NMDA-based pattern discrimination in a modeled cortical neuron”. *Neural Computation*. 4(4): 502–517.

[161] Mendat, D. R., S. Chin, S. Furber, and A. G. Andreou. 2015. “Markov Chain Monte Carlo inference on graphical models using event-based processing on the SpiNNaker neuromorphic architecture”. In: *2015 49th Annual Conference on Information Sciences and Systems (CISS)*. 1–6. DOI: 10.1109/CISS.2015.7086903.

[162] Merolla, P. A., J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. 2014. “A million spiking-neuron integrated circuit with a scalable communication network and interface”. *Science*. 345(6197): 668–673. DOI: 10.1126/science.1254642. eprint: http://www.sciencemag.org/content/345/6197/668.full.pdf.

[163] Merolla, P. A., J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. 2014. “A million spiking-neuron integrated circuit with a scalable communication network and interface”. *Science*. 345(6197): 668–673. ISSN: 0036-8075. DOI: 10.1126/science.1254642.

[164] Mikaitis, M., D. R. Lester, D. Shang, S. Furber, G. Liu, J. Garside, S. Scholze, S. Höppner, and A. Dixius. 2018. “Approximate fixed-point elementary function accelerator for the SpiNNaker-2 neuromorphic chip”. In: *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*. 37–44. DOI: 10.1109/ARITH.2018.8464785.

[165] Mikaitis, M., P. G. Garibaldi, J. C. Knight, and S. B. Furber. 2018. “Neuromodulated synaptic plasticity on the SpiNNaker neuromorphic system”. *Frontiers in Neuroscience*. 12(February): 1–13. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00105.

[166] Mikolov, T., M. Karafiát, L. Burget, J. Cernocky, and S. Khudanpur. 2010. “Recurrent neural network based language model.” In: *Interspeech*. Vol. 2. 3.

[167] Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. 2016. “Asynchronous methods for deep reinforcement learning”. 48. arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[168] Moore, G. E. 1965. “Craming more components onto integrated circuits”. *Electronics*. 38(8): 114–117. ISSN: 0018-9219. DOI: 10.1109/jproc.1998.658762.

[169] Moore, S., P. Fox, S. Marsh, A. Markettos, and A. Mujumdar. 2012. “Bluehive – A field-programmable custom computing machine for extreme-scale real-time neural network simulation”. In: *2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 133–140. DOI: 10.1109/FCCM.2012.32.

[170] Morrison, A., A. D. Aertsen, M. Diesmann, A. Morrison, and M. Diesmann. 2007. “Spike-timing-dependent plasticity in balanced random networks”. *Neural Computation Massachusetts Institute of Technology*. 19: 1437–1467. ISSN: 0899-7667. DOI: 10.1162/neco.2007.19.6.1437.

[171] Morrison, A., M. Diesmann, and W. Gerstner. 2008. “Phenomenological models of synaptic plasticity based on spike timing”. *Biological Cybernetics*. 98(6): 459–478. ISSN: 03401200. DOI: 10.1007/s00422-008-0233-1.

[172] Morrison, A., C. Mehring, T. Geisel, A. D. Aertsen, and M. Diesmann. 2005. “Advancing the boundaries of high-connectivity network simulation with distributed computing.” *Neural Computation*. 17(8): 1776–1801. ISSN: 0899-7667. DOI: 10.1162/0899766054026648.

[173] Muller, J.-M. 2016. *Elementary Functions – Algorithms and Implementation*. 3rd ed. Birkhäuser Basel.

[174] Mundy, A., J. Heathcote, and J. D. Garside. 2016. “On-chip order-exploiting routing table minimization for a multicast supercomputer network”. *IEEE International Conference on High Performance Switching and Routing, HPSR*. 2016 July: 148–154. ISSN: 23255609. DOI: 10.1109/HPSR.2016.7525659.

[175] Natrella, M. 2010. “NIST/SEMATECH e-handbook of Statistical Methods”. Ed. by C. Croarkin and P. Tobias. https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc431.htm. (Accessed on 2018).

[176] Neil, D. and S.-C. Liu. 2014. “Minitaur, an event-driven FPGA-based spiking network accelerator”. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 22(12): 2621–2628. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2013.2294916.

[177] Neumarker, F., S. Höppner, A. Dixius, and C. Mayr. 2016. “True random number generation from bang-bang ADPLL jitter”. In: *2016 IEEE Nordic Circuits and Systems Conference (NORCAS)*. 1–5. DOI: 10.1109/NORCHIP.2016.7792875.

[178] Neuroinformatics of the University of Zürich, I. of. 2007. “jAER: Java tools for Address-Event Representation (AER) neuromorphic vision and audio sensor processing”. URL: https://github.com/SensorsINI/jaer (accessed on 2018).

[179] Newcomer, J. W., N. B. Farber, and J. W. Olney. 2000. “NMDA receptor function, memory, and brain aging”. *Dialogues in Clinical Neuroscience*. 2(3): 219.

[180] Nguyen, T. 2013. “Total number of synapses in the adult human neocortex”. *Undergraduate Journal of Mathematical Modeling: One + Two*. 3(1). DOI: 10.5038/2326-3652.3.1.26.

[181] Noethen, B., O. Arnold, E. Perez Adeva, T. Seifert, E. Fischer, S. Kunze, E. Matus, G. Fettweis, H. Eisenreich, G. Ellguth, S. Hartmann, S. Höppner, S. Schiefer, J.-U. Schlusler, S. Scholze, D. Walter, and R. Schüffny. 2014. “A 105GOPS 36mm2 heterogeneous SDR MPSoC with energy-aware dynamic scheduling and iterative detection-decoding for 4G in 65nm CMOS”. In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International.* DOI: 10.1109/ISSCC.2014.6757394.

[182] Nordlie. 2009. “Visualizing neuronal neonnectivity with connectivity pattern tables twork c”. *Frontiers in Neuroinformatics*. 3(January): 1–15. ISSN: 16625196. DOI: 10.3389/neuro.11.039.2009.

[183] O’Connor, P., D. Neil, S.-C. Liu, T. Delbruck, and M. Pfeiffer. 2013. “Real-time classification and sensor fusion with a spiking deep belief network”. *Frontiers in Neuroscience*. 7(178). ISSN: 1662-453X. DOI: 10.3389/fnins.2013.00178.

[184] Oja, E. 1983. “A simplified neuron model as a principal component analyzer”. *Journal of Mathematical Biology*. 15: 267–273.

[185] Orchard, G., X. Lagorce, C. Posch, S. B. Furber, R. Benosman, and F. Galluppi. 2015. “Real-time event-driven spiking neural network object recognition on the SpiNNaker platform”. In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2413–2416. DOI: 10.1109/ISCAS.2015.7169171.

[186] Painkras, E., L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber. 2013. “SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation”. *IEEE Journal of Solid-State Circuits*. 48(8): 1943–1953. ISSN: 00189200. DOI: 10.1109/JSSC.2013.2259038.

[187] Pakkenberg, B., D. Pelvig, L. Marner, M. J. Bundgaard, H. J. G. Gundersen, J. R. Nyengaard, and L. Regeur. 2003. “Aging and the human neocortex”. *Experimental Gerontology*. 38(1-2): 95–99. ISSN: 05315565. DOI: 10.1016/S0531-5565(02)00151-1.

[188] Partzsch, J., S. Höppner, M. Eberlein, R. Schüffny, C. Mayr, D. R. Lester, and S. Furber. 2017. “A fixed point exponential function accelerator for a neuromorphic many-core system”. In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4. DOI: 10.1109/ISCAS.2017.8050528.

[189] Perea, G., M. Navarrete, and A. Araque. 2009. “Tripartite synapses: Astrocytes process and control synaptic information”. *Trends in Neurosciences*. 32(8): 421–431. ISSN: 0166-2236. DOI: 10.1016/j.tins. 2009.05.001.

[190] Pérez-Carrasco, J. A., B. Zhao, C. Serrano, B. Acha, T. Serrano-Gotarredona, S. Chen, and B. Linares-Barranco. 2013. “Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward ConvNets”. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 35(11): 2706–2719. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2013.71.

[191] Pfeil, T., T. Potjans, S. Schrader, W. Potjans, J. Schemmel, M. Diesmann, and K. Meier. 2012. “Is a 4-Bit synaptic weight resolution enough? – constraints on enabling spike-timing dependent plasticity in neuromorphic hardware”. *Frontiers in Neuroscience*. 6: 90. ISSN: 1662-453X. DOI: 10.3389/fnins.2012.00090.

[192] Pfister, J.-P. 2006. “Triplets of spikes in a model of spike timing-dependent plasticity”. *Journal of Neuroscience*. 26(38): 9673–9682. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.1425-06.2006.

[193] Pineda-Garcıéa, G. 2019. “A Visual Pipeline Using Networks of Spiking Neurons”. *PhD Thesis*. The University of Manchester. 166.

[194] Pineda-Garcıéa, G., P. Camilleri, Q. Liu, and S. Furber. 2016. “pyDVS: An extensible, real-time Dynamic Vision Sensor emulator using off-the-shelf hardware”. In: *IEEE Symposium Series on Computational Intelligence, SSCI*. ISBN: 9781509042401. DOI: 10.1109/SSCI.2016.7850249.

[195] Plana, L. A., D. Clark, S. Davidson, S. Furber, J. Garside, E. Painkras, J. Pepper, S. Temple, and J. Bainbridge. 2011. “SpiNNaker: Design and Implementation of a GALS Multicore System-on-Chip”. *ACM J. Emerg. Tech. Comput.* 7(4): 17:1–17:18.

[196] Plana, L. A., S. B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang. 2007. “A GALS infrastructure for a massively parallel multiprocessor”. *IEEE Design Test of Computers*. 24(5): 454–463. ISSN: 0740-7475. DOI: 10.1109/MDT.2007.149.

[197] Plana, L. A., J. Heathcote, J. S. Pepper, S. Davidson, J. Garside, S. Temple, and S. B. Furber. 2014. “spI/O: A library of FPGA designs and reusable modules for I/O in SpiNNaker systems”. DOI: 10.5281/zenodo.51476.

[198] Plana, L. A. 2017. “Interfacing AER devices to SpiNNaker using an FPGA”. *Tech. Rep.* http://spinnakermanchester.github.io/docs/spinn-app-8.pdf, SpiNNaker Application Note 8. University of Manchester.

[199] Plesser, H. E., J. M. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig. 2007. “Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers”. In: *Euro Par 2007 Parallel Processing*. Ed. by A.-M. Kermarrec, L. Bougé, and T. Priol. Berlin, Heidelberg: Springer Berlin Heidelberg. 672–681. ISBN: 978-3-540-74466-5.

[200] Ponulak, F. and A. Kasinski. 2010. “Supervised learning in spiking neural networks with ReSuMe: Sequence learning, classification, and spike shifting”. *Neural Computation*. 22(2): 467–510.

[201] Potjans, T. C. and M. Diesmann. 2014. “The cell-type specific cortical microcircuit: Relating structure and activity in a full-scale spiking network model”. *Cerebral Cortex*. 24(3): 785–806. ISSN: 10473211. DOI: 10.1093/cercor/bhs358.

[202] Prescott, T. J., F. M. Montes González, K. Gurney, M. D. Humphries, and P. Redgrave. 2006. “A robot model of the basal ganglia: Behavior and intrinsic processing”. *Neural Networks*. 19(1): 31–61. ISSN: 08936080. DOI: 10.1016/j.neunet.2005.06.049.

[203] Radford, A., L. Metz, and S. Chintala. 2015. “Unsupervised representation learning with deep convolutional generative adversarial networks”. *arXiv preprint arXiv:1511.06434*.

[204] Ramón y Cajal, S. 1928. *Degeneration & Regeneration of the Nervous System*. London: Oxford University Press, Humphrey Milford.

[205] Rast, A. D., A. B. Stokes, S. Davies, S. V. Adams, H. Akolkar, D. R. Lester, C. Bartolozzi, A. Cangelosi, and S. Furber. 2015. “Transport-Independent Protocols for Universal AER Communications”. In: *Neural Information Processing*. Ed. by S. Arik, T. Huang, W. K. Lai, and Q. Liu. Cham: Springer International Publishing. 675–684. ISBN: 978-3-319-26561-2.

[206] Rauch, A., G. La Camera, H.-R. Lüscher, W. Senn, and S. Fusi. 2003. “Neocortical pyramidal cells respond as integrate-and-fire neurons to in vivo-like input currents”. *Journal of Neurophysiology*. 90(3): 1598–1612.

[207] Rhodes, O., A. Bogdan, C. Brenninkmeijer, S. Davidson, D. Fellows, A. Gait, D. R. Lester, M. Mikaitis, L. A. Plana, A. G. D. Rowley, A. B. Stokes, and S. B. Furber. 2018. “sPyNNaker: A software package for running PyNN simulations on SpiNNaker”. *Frontiers in Neuroscience*. 12 (November). ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00816.

[208] Rhodes, O., P. A. Bogdan, C. Brenninkmeijer, S. Davidson, D. Fellows, A. Gait, D. R. Lester, M. Mikaitis, L. A. Plana, A. G. D. Rowley, A. B. Stokes, and S. B. Furber. 2018. “Supplementary material: sPyNNaker: A software package for running PyNN simulations on SpiNNaker”. *Frontiers. Collection.*

[209] Richter, C., S. Jentzsch, R. Hostettler, C. Richter, S. Jentzsch, R. Hostettler, F. Röhrbein, P. van der Smagt, and J. Conradt. 2016. “Musculoskeletal Robots: Scalability in Neural Control”. *IEEE Robotics & Automation Magazine*. 23(4): 128–137. DOI: 10.1109/MRA.2016. 2535081.

[210] Rodriguez-Pineda, J. A. 2000. “Competitive Hebbian Learning Through Spiking-Timing Dependent Plasticity (STDP)”. *Thesis & Dissertation*. 3: 919–926. ISSN: 1308-0911. DOI: 10.16953/deusbed.74839.

[211] Roelfsema, P. R. and A. v. Ooyen. 2005. “Attention-gated reinforcement learning of internal representations for classification”. *Neural Computation*. 17(10): 2176–2214.

[212] Rotter, S. and M. Diesmann. 1999. “Exact digital simulation of time-invariant linear systems with applications to neuronal modeling.” *Biological Cybernetics*. 81(5-6): 381–402. ISSN: 0340-1200. DOI: 10.1007/s004220050570.

[213] Rowley, A. G. D., C. Brenninkmeijer, S. Davidson, D. Fellows, A. Gait, D. R. Lester, L. A. Plana, O. Rhodes, A. B. Stokes, and S. Furber. 2019. “SpiNNTools: The execution engine for the SpiNNaker platform”. *Frontiers in Neuroscience*. 13: 231. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00231.

[214] Rubin, J., D. D. Lee, and H. Sompolinsky. 2001. “Equilibrium properties of temporally asymmetric Hebbian plasticity”. *Physical Review Letters*. 86(2): 364–367. ISSN: 00319007. DOI: 10.1103/PhysRevLett. 86.364. arXiv: 0007392 [cond-mat].

[215] Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.* 2015. “ImageNet large scale visual recognition challenge”. *International Journal of Computer Vision*. 115(3): 211–252.

[216] Schraut, F., S. Höppner, C. Mayr, and H. Eisenreich. 2019. “A Fast Lock-in Ultra Low-Voltage ADPLL Clock Generator with Adaptive Body Biasing in 22nm FDSOI Technology”. In: *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*.

[217] Sen-Bhattacharya, B., S. Member, S. James, O. Rhodes, I. Sugiarto, A. B. Stokes, K. Gurney, A. Rowley, A. B. Stokes, K. Gurney, and S. Furber. 2018. “Building a spiking neural network model of the basal ganglia on SpiNNaker”. *IEEE Transactions on Cognitive and Developmental Systems*: 1–1. ISSN: 2379-8920. DOI: 10.1109/TCDS.2018.2797426.

[218] Serrano-Gotarredona, R., M. Oster, P. Lichtsteiner, A. Linares-Barranco, R. Paz-Vicente, F. Gomez-Rodriguez, L. Camuñas-Mesa, R. Berner, M. Rivas-Pérez, T. Delbruck, S. Liu, R. Douglas, P. Hafliger, G. Jiménez-Moreno, A. C. Ballcels, T. Serrano-Gotarredona, A. J. Acosta-Jiménez, and B. Linares-Barranco. 2009. "CAVIAR: A 45k Neuron, 5M Synapse, 12G connects/s AER hardware sensory-processing-learning-actuating system for high-speed visual object recognition and tracking". *IEEE Transactions on Neural Networks*. 20(9): 1417–1438. ISSN: 1045-9227. DOI: 10.1109/TNN.2009.2023653.

[219] Serrano-Gotarredona, T. and B. Linares-Barranco. 2013. "A 128×128 1.5% contrast sensitivity 0.9% FPN 3 μs latency 4 mW asynchronous frame-free dynamic vision sensor using transimpedance preamplifiers". *IEEE Journal of Solid-State Circuits*. 48(3): 827–838. ISSN: 0018-9200. DOI: 10.1109/JSSC.2012.2230553.

[220] Serrano-Gotarredona, T., A. G. Andreou, and B. Linares-Barranco. 1999. "AER image filtering architecture for vision-processing systems". *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*. 46(9): 1064–1071.

[221] Shannon, C. E. 1948. "A mathematical theory of communication". *The Bell System Technical Journal*. 27(3): 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[222] Sharp, T., F. Galluppi, A. Rast, and S. Furber. 2012. "Power-efficient simulation of detailed cortical microcircuits on SpiNNaker". *Journal of Neuroscience Methods*. 210(1): 110–118. ISSN: 01650270. DOI: 10.1016/j.jneumeth.2012.03.001.

[223] Sharp, T., L. A. Plana, F. Galluppi, and S. Furber. 2011. "Eventdriven simulation of arbitrary spiking neural networks on SpiNNaker". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7064 LNCS. *3*. Heidelberg. 424–430. ISBN: 9783642249648. DOI: 10.1007/978-3-642-24965-5_48.

[224] Shi, Y., S. B. Furber, J. Garside, and L. A. Plana. 2009. "Fault tolerant delay insensitive inter-chip communication". In: *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*. 77–84. DOI: 10. 1109/ASYNC.2009.21.

[225] Shi, Y. 2010. "Fault-Tolerant Delay-Insensitive Communication". *PhD Thesis*. The University of Manchester. URL: http://apt.cs.manchester.ac.uk/publications/thesis/YShi10_phd.php.

[226] Siegert, A. J. 1951. "On the first passage time probability problem". *Physical Review*. 81(4): 617.

[227] Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. 2016. "Mastering the game of Go with deep neural networks and tree search". *Nature*. 529(7587): 484–489. ISSN: 0028-0836. DOI: 10.1038/nature16961.

[228] Simonyan, K. and A. Zisserman. 2014. "Very deep convolutional networks for large-scale image recognition". *arXiv preprint arXiv:1409.1556*.

[229] Sjöström, P. J., G. G. Turrigiano, and S. B. Nelson. 2001. "Rate, timing, and cooperativity jointly determine cortical synaptic plasticity". *Neuron*. 32(6): 1149–1164. ISSN: 08966273. DOI: 10.1016/S0896-6273(01)00542-6.

[230] Sloss, A., D. Symes, and C. Wright. 2004. *ARM System Developer's Guide: Designing and Optimizing System Software*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1558608745.

[231] Sofroniew, M. V. and H. V. Vinters. 2010. "Astrocytes: Biology and pathology". *Acta Neuropathologica*. 119(1): 7–35. ISSN: 00016322. DOI: 10.1007/s00401-009-0619-8. arXiv: NIHMS150003.

[232] Softky, W. R. and D. Koch. 2015. "The highly irregular temporal integration firing of cortical cells is inconsistent of random EPSPs". *The Journal of*. 13(January): 334–350.

[233] Song, S. and L. F. Abbott. 2001. "Cortical development and remapping through spike timing-dependent plasticity". *Neuron*. 32(2): 339–350. ISSN: 08966273. DOI: 10.1016/S0896-6273(01)00451-2.

[234] SpiNNaker. 2011. "SpiNNaker Application Programming Interface". *Tech. Rep.* University of Manchester. URL: http://spinnakermanchester.github.io/docs/SpiNNapi_docV200.pdf.

[235] SpiNNaker. 2011. "SpiNNaker Datasheet Version 2.02". *Tech. Rep.* University of Manchester. URL: http://spinnakermanchester.github.io/docs/SpiNN2DataShtV202.pdf.

[236] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. 2014. "Dropout: A simple way to prevent neural networks from overfitting". *Journal of Machine Learning Research*. 15: 1929–1958. ISSN: 15337928. DOI: 10.1214/12-AOS1000. arXiv: 1102.4807.

[237] Stanley, K. O. and R. Miikkulainen. 2002. "Evolving Neural Networks through Augmenting Topologies". *Evolutionary Computation*. 10(2): 99–127. ISSN: 1063-6560, 1530-9304. DOI: 10.1162/106365602320169811.

[238] Stein, R. B. 1967. "Some models of neuronal variability". *Biophysical Journal*. 7(1): 37–68.

[239] Stromatias, E., D. Neil, F. Galluppi, M. Pfeiffer, S. Liu, and S. Furber. 2015. “Live demonstration: Handwritten digit recognition using spiking deep belief networks on SpiNNaker”. In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1901–1901. DOI: 10.1109/ISCAS.2015.7169034.

[240] Stromatias, E., D. Neil, F. Galluppi, M. Pfeiffer, S. Liu, and S. Furber. 2015. “Scalable energy-efficient, low-latency implementations of trained spiking Deep Belief Networks on SpiNNaker”. In: *2015 International Joint Conference on Neural Networks (IJCNN)*. 1–8. DOI: 10.1109/IJCNN.2015.7280625.

[241] Stromatias, E. 2016. “Scalability and robustness of artificial neural networks”. *PhD Thesis*. University of Manchester.

[242] Stromatias, E., F. Galluppi, C. Patterson, and S. Furber. 2013. “Power analysis of large-scale, real-time neural networks on SpiNNaker”. In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 1–8.

[243] Stromatias, E., D. Neil, M. Pfeiffer, F. Galluppi, S. Furber, and S.-C. Liu. 2015. “Robustness of spiking Deep Belief Networks to noise and reduced bit precision of neuro-inspired hardware platforms”. *Frontiers in Neuroscience*. 9: 222. ISSN: 1662-453X. DOI: 10.3389/fnins.2015. 00222.

[244] Sugiarto, I., L. A. Plana, S. Temple, B. S. Bhattacharya, S. Furber, and P. Camilleri. 2019. “Profiling a many-core neuromorphic platform”. *2017 IEEE 11th International Conference on Application of Information and Communication Technologies (AICT)*: 1–6. DOI: 10.1109/icaict.2017.8687014.

[245] Sumner, C. J., E. A. Lopez-Poveda, L. P. O’Mard, and R. Meddis. 2002. “A revised model of the inner-hair cell and auditory-nerve complex”. *The Journal of the Acoustical Society of America*. 111(5): 2178–2188.

[246] Surungan, T., F. P. Zen, and A. G. Williams. 2015. “Spin glass behavior of the antiferromagnetic Heisenberg model on scale free network”. *Journal of Physics: Conference Series*. 640(1): 012005. URL: http://stacks.iop.org/1742-6596/640/i=1/a=012005.

[247] Sutskever, I., O. Vinyals, and Q. V. Le. 2014. “Sequence to sequence learning with neural networks”. In: *Advances in Neural Information Processing Systems*. 3104–3112.

[248] Sutton, R. S. and A. G. Barto. 1998. *Reinforcement Learning: An Introduction*. The MIT Press.

[249] Szeliski, R. 2010. *Computer Vision: Algorithms and Applications*. Springer Science & Business Media.

[250] Telecommunication Industry Association. 2006. “ANSI/TIA-942: Data Center Standards”.

[251] Temple, S. 2016. "SARK – SpiNNaker Application Runtime Kernel". *Tech. Rep.* University of Manchester. URL: http://spinnakermanchester.github.io/docs/sarkV200.pdf.

[252] Terreros, G. and P. H. Delano. 2015. "Corticofugal modulation of peripheral auditory responses". *Frontiers in Systems Neuroscience*. 9.

[253] Tewari, S. G. and K. K. Majumdar. 2012. "A mathematical model of the tripartite synapse: astrocyte-induced synaptic plasticity". *Journal of Biological Physics*. 38(3): 465–496. ISSN: 1573-0689. DOI: 10.1007/s10867-012-9267-7.

[254] Theodosis, D. T., D. A. Poulain, and S. H. R. Oliet. 2008. "Activity-dependent structural and functional plasticity of astrocyte-neuron interactions". *Physiological Reviews*. 88(3): 983–1008. ISSN: 0031-9333. DOI: 10.1152/physrev.00036.2007. eprint: http://physrev.physiology.org/content/88/3/983.full.pdf.

[255] THOCP. 2013. "Cray 1, Available online". URL: http://www.thocp.net/hardware/cray_1.htm.

[256] Turing, A. 1950. "Computing machinery and intelligence". *Mind – A Quarterly Review of Psychology and Philosophy*. 59(236): 433–460. DOI: https://doi.org/10.1093.

[257] Turing, A. 1936. "On computable numbers, with an application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society*. 42(1): 230–265. DOI: 10.2307/2268810.

[258] Vainbrand, D. and R. Ginosar. 2010. "Network-on-Chip architectures for neural networks". In: *2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*. 135–144. DOI: 10.1109/NOCS.2010.23.

[259] Vandesompele, A., F. Walter, and R. Florian. 2016. "Neuro-evolution of spiking neural networks on SpiNNaker neuromorphic hardware". In: *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. 1–6. DOI: 10.1109/SSCI.2016.7850250.

[260] Vinyals, O., I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver. 2019. "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II". https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii.

[261] Vogels, T. P., H. Sprekeler, F. Zenke, C. Clopath, and W. Gerstner. 2011. "Inhibitory plasticity balances excitation and inhibition in sensory pathways and memory networks." *Science (New York, N.Y.)* 334(6062): 1569–1573. ISSN: 1095-9203. DOI: 10.1126/science.1211095. arXiv: 20.

[262] Vogels, T. P. and L. F. Abbott. 2005. "Signal propagation and logic gating in networks of integrate-and-fire neurons". *Journal of Neuroscience*. 25(46): 10786–10795. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.3508-05.2005. arXiv: NIHMS150003.

[263] Vogginger, B., R. Schüffny, A. Lansner, L. Cederström, J. Partzsch, and S. Höppner. 2015. "Reducing the computational footprint for real-time BCPNN learning". *Frontiers in Neuroscience*. 9: 2. ISSN: 1662-453X. DOI: 10.3389/fnins.2015.00002.

[264] von der Malsburg, C. 1973. "Self-organization of orientation sensitive cells in the striate cortex". *Kybernetik*. 14(2): 85–100. ISSN: 1432-0770. DOI: 10.1007/BF00288907.

[265] Westerman, L. A. and R. L. Smith. 1987. "Conservation of adapting components in auditory-nerve responses". *The Journal of the Acoustical Society of America*. 81(3): 680–691.

[266] Yan, Y., D. Kappel, F. Neumaerker, J. Partzsch, B. Vogginger, S. Hoeppner, S. Furber, W. Maass, R. Legenstein, and C. Mayr. 2019. "Efficient reward-based structural plasticity on a SpiNNaker 2 prototype". *IEEE Transactions on Biomedical Circuits and Systems*: 1–1. ISSN: 1932-4545. DOI: 10.1109/TBCAS.2019.2906401.

[267] Furber, S.B., D.A. Edwards, and J.D. Garside. 2000. "AMULET3: a 100 MIPS Asynchronous Embedded Processor". In *Proceedings 2000 International Conference on Computer Design*. pp. 329–334.

Index

2-of-7 code, 37, 38, 62, 63, 164
22FDX, 265, 277, 278
2D grid, 71
2D mesh network, 268
3-of-6, 37
3D lattice, 159
3D torus, 49
3D-printed, 133
A3C, 162
ABB, xiii, 277, 279
accuracy, 110
ack, 64, 65
Acorn Computers, 4
action selection, 144–146
activation function, 193, 194, 202, 255
actuator, 129
AD, xiii, 240, 241, 243, 244
additive weight dependence, 222
Address Event Representation, *see* AER
ADPLL, xiii, 273
AER, xiii, 6–10, 19, 22, 55, 107, 164, 166
AGI, xiii, 128
AHB, xiii, 27, 28, 34, 39, 275
AI, xiii, 128, 155, 156, 160, 161, 163, 179, 203, 263
aiCTX, 263
AlexNet, 162
AlphaGo, 160
Altera, 16
AMBA, 27
Amulet, 5
Amulet3, 14
AN, xiii, 139–142
analytical response function, 187, 189
ANN, xiv, 161, 178–181, 188, 190, 191, 193–196, 199–201, 203, 204, 246, 255, 257, 260–262, 264
antiferromagnetic, 158, 159
API, xiv, 104, 106, 118, 122
Application Edge, 86, 101
Application Graph, 86
Application Vertex, 86, 101
architecture, 271
area under curve, 175
ARM, xiv, 4, 5, 7, 9, 10, 14, 16, 18, 19, 25, 27, 28, 30, 34, 39, 41, 43, 44, 48, 49, 51, 54, 55, 57, 67, 68, 73, 76, 80, 120, 127, 172, 176, 207, 216, 217, 263, 266, 270, 272, 275–277
armcc, 80
Artificial General Intelligence, *see* AGI
artificial intelligence, 4

ASIC, xiv, 18–21, 25, 27, 43–45, 264
associative memory, 5, 8
astrocyte, 231
asynchronous advantage actor-critic, 162
asynchronous handshake, 37, 38, 66
asynchronous interconnect, 35, 38, 39, 54
asynchronous interconnection technology, 38
asynchronous link, 43, 50
asynchronous logic, 5, 7
asynchronous network, 44, 45
asynchronous regime, 221
auditory modelling, 138
Auditory Nerve, *see* AN
auditory nerve fibre, 138, 141
auditory pathway, 137, 138, 143
autoencoders, 161
AXI, xiv, 28, 39
backpropagation, 178, 196, 261
bandwidth, 10, 13, 19, 20, 31, 48, 50, 62, 66, 95, 97, 114, 122, 133, 265, 267
Basilar Membrane, 139
BBC Microcomputer, 4
BCM, 207
BCPNN, 274
BG, xiv, 128, 132, 143–146
BigCrush, 273
biological evidence, 235
bisection bandwidth, 50, 51, 62
blacklist, 54, 55, 60, 61, 79, 83
blinkenlights, 43, 80
blob, 135, 136
BMP, xiv, 57, 59–61, 66–68, 74
Board Management Processor, *see* BMP
Board-to-board latency, 66
Boolean, 41, 44
boot, 30, 38
boot image, 81
boot sequence, 43
bottleneck, 48
boxplot, 254
brain, 1, 5
BrainChip, 263
BrainScaleS, 207
brainstem, 144
brainstem reticular formation, 144
Brian, 103, 171, 176
buffer, 270
bug, 248
C, 91, 92, 99, 104, 118, 119, 124
C++, 97
C2C, 269, 270
CA, 174
cable installation, 72
cache, 5
CAD, 65
Cadence, 263
callback, 81, 92, 107–109, 112, 114, 122, 123
callback priority, 81
callback queue, 81
CAM, xiv, 22, 24
CAN, xiv, 68
Cartesian, 20
CD, 170
centre-surround filtering, 135
centre-surround kernel, 246
chip area, 273
chip-to-chip links, 267
classification, 166, 254
classification accuracy, 174–177, 199–202, 255
classification latency, 176, 177
clock cycle, 42
clue, 155
cluster, 203
CMOS, xv, 16, 45, 51, 277
cochlea, xvii, 138, 139
cochlea model, 138
coding theory, 155

communication, 266
communication network, 43
competition, 155
compiler, 80, 274
compression, 95
computable, 3
computation, 189
Computational Neuroscience, 78, 84
Computer Science, 3, 7, 9
computer vision, 133
Content-Addressable Memory, *see* CAM
contiguous data, 39
Contrastive Divergence, *see* CD
control system, 130
control theory, 129
Controller Area Network, *see* CAN
convergence, 154
ConvNet, xv, 161–166, 169, 196, 197, 201, 264
convolutional filter, 259
convolutional layer, 163, 164, 168, 169
Conway, John, 99–102
cortex, 144
Cortex-M4, 270
cortical regions, 138
counter, 43, 44
CPU, xv, 88, 127, 141, 142, 217
CRC, xv, 30, 46, 64–66
credit assignment, 234, 236
cross-correlation, 135
crossover, 256
crystal, 156
CSP, xv, 148–154, 156, 158
current, 181
Cyclic-Redundancy Check, *see* CRC
data processing, 163
data-management, 103
DBN, xv, 160, 169–178
dBSPL, 141, 142
DC, xv, 56
DDR, xv, 43
deadlock, 9
debug, 248
debugging, 43
decay rate, 233
decision making, 146
Deep Belief Network, *see* DBN
deep learning, 160–162, 178
deep neural network, 169
DeepMind, 160, 162
default routeing, 9
defautDMA, 272
delay-insensitive code, 62
Delay-Locked Loop, *see* DLL
depression, 211
design space exploration, 277
DfT, xv, 267
diagnostic, 72
die area, 51
Dieharder, 273
digital signal processing, 138, 263
Dirac, 171, 210, 233
Dirac delta, 171, 210, 233, 234
Direct Memory Access, *see* DMA
direction, 101, 252
discriminator, 162
dispatcher thread, 108
distributed system, 77
DLL, xv, 54
DMA, xv, 10, 25, 27–30, 39, 42, 46, 48, 51, 81, 107, 108, 112, 114, 116–118, 123, 213, 215, 267, 272
DNN, xv, 161, 200, 263, 275, 276
dopamine, 146, 232, 233, 235
DOTA 2, 160
DRAM, 10
DRNL, xv, 139, 140
dropped packets, 47
DSI, xv, 251–254

DTCM, xvi, 27, 29, 82, 88, 106–108, 112–114, 122, 123, 165, 166, 214, 217
DVFS, xvi, 279
DVS, xvi, 55, 57, 164, 166, 170, 171
Dynamic Vision Sensor, *see* DVS
EA, xvi, 255–257, 260, 261
edge detection, 135
EIEIO packet, 98
EIEIO protocol, 97
elementary motion decomposition, 238, 249
eligibility trace, 206, 232, 233, 235
emergency routeing, 34, 35, 44, 49
energy consumption, 141
energy consumption auditory model, 141
energy efficiency, 74, 261, 277
Engineering, 147
entropy, 155, 156, 158, 251–254, 273
entropy extraction, 273
Entschedungsproblem, 3
EoP, xvi, 35–38
EPSRC, xii, xvi, 5, 16
error backpropagation, 261
error correction, 155
error detection, 155
ES, xvi, 260
escargot, 155
Eta Compute, 263
Ethernet, 29, 30, 42, 43, 55–57, 59, 60, 67, 79, 80, 82, 83, 95–97, 99, 105, 122, 257, 265
EU, 76
Event Camera, 55
event-based, 219
event-based computation, 133
event-based operating system, *see* SpiNNaker, SpiN1API, 108, 118, 122
event-based processing, 133
event-driven, 42, 55, 56, 105, 107, 140, 165, 178, 213, 234, 249
event-driven clustering algorithm, 164
event-driven computation, 234
event-driven library, 105, 107
excitatory projection, 104
executable binary, 86
exhibition, 132
exponential, 150, 179, 208, 226, 274
exponential decay, 274
exponential function, 208
F-score, 252
fabrication cost, 50
Fast Interrupt Requests, *see* FIQ
fault, 37
fault tolerance, 79
FDSOI, 265, 277
feature map, 163, 169
Feature maps, 165
feedback, 132, 143
ferromagnetic, 158
ferromagnetic lattice, 159
field-effect transistors, xv
Field-Programmable Gate Array, *see* FPGA
FIFO, xvi, 65, 267
filter bank, 139
FIQ, xvi, 30, 81, 108, 112, 118
FIQ thread, 108, 112
firing rate, 180, 191, 194, 198, 199
fixed simulation timestep, 109
fixed-point datatypes, 121
Flash, xvi
flash, 55
Flash memory, 55
flit, xvi, 35–38
FPGA, xiv, xvi, xxiii, 55, 57–60, 62, 63, 65, 66, 74, 178
FPU, xvi, 172, 270
frequency, 138, 139
fully connected layer, 164

fully-depleted silicon on insulator, *see* FDSOI
Furhat, 129
GA, xvii, 256, 258–260
Gabor, 134
Gabor filter, 134, 135, 164
GALS, xvii, 267
Game of Life, 99, 102
GAN, xvii, 162
ganglion cells, 133
gate-level netlist, 277
Gaussian, 135, 182, 188
Gaussian distribution, 182, 237
Gaussian kernel, 135
gcc, 80
general purpose I/O, 43
generalisation, 178, 203
Generative Adverserial Network, *see* GAN
generator, 162
genetic computer, 148
GlobalFoundries, 265
glutamatergic, 144
GND, 56, 58
GNU, xvii, 80
Go, 160
Google, 263
GoogLeNet, 162
GPU, xvii, 263
gradient-free optimisation methods, 255
graph colouring, 152, 153
Graphcore, 263
Graphical User Interface, *see* GUI
GUI, xvii, 72
h-test, 252
hardware, 32, 44, 47, 48, 69, 77, 84, 141–143, 147, 179, 203, 204, 213, 274
Hardware Abstraction Layer, 81
hardware event, 81, 107, 108, 116
HBP, xvii, 76, 103, 207, 219
Hebb, 206
Hebbian, 15, 231
hidden layer, 171, 173, 175, 177, 258
High-Speed Serial Link, *see* HSSL
histogram, 44, 141, 177, 248, 253
host computer, 105
host machine, 95
HPC, xvii, 9, 12, 207
HSSL, xvii, 63–65
Huawei, 263
Hubel, 161
Human Brain Project, *see* HBP
hyperparameters, 261
hypersurface, 148
I/O, 27, 43, 51, 55, 66, 265, 267, 268
I/O library, 55
IHC, xvii, 138–141
image recognition, 162
image subsampling, 163
ImageNet, 162
Imagination, 263
inference, 264
infinite impulse response filters, 139
information theory, 238
inhibitory plasticity, 222–224
inhibitory projection, 104
inhibitory synapses, 250
Inner Hair Cell, *see* IHC
input current, 118–120, 149, 171, 181, 183, 189, 190, 192, 229
Institute of Neuroinformatics, 170
Instruction tightly-coupled memory, *see* ITCM
integrate-and-fire, 226
integrated circuit, 40
Intel, 143, 263
Intellectual Property, 50
inter-board interconnect, 63
interface, 43

interrupt, 42
intractable, 147, 148, 206
IP, xvii, 54, 55, 66, 68, 80, 81, 88
ISI, xvii, 216
Ising, 152, 156–158
Ising spin glasses, 152
ITCM, xviii, 27, 29, 80–82, 106
Izhikevich, 118, 121, 144, 226, 227, 229, 273
jAER, 166
Java, 97
Jenkins, 76
Joint Test Action Group, *see* JTAG
JTAG, xviii, 43, 59
Jupyter, 248
kernel, 42, 118, 134, 135, 164, 165, 168, 172, 197–199, 211, 222, 246
Kilburn, Tom, 3
Kinect, 129
KISS64, 273
KISS99, 273
Kronecker delta, 227
L2L, xviii, 261
latency, 274
Latin square, 155
Latin squares, 152
layer, 163
LCD, xviii, 68
leakage power consumption, 277
learning rule, 9, 178, 219, 222, 227, 232, 234, 274
LED, xviii, 55, 59, 72
Lee Sedol, 160
LG, 263
LIF, xviii, 106, 118, 120, 121, 149, 170, 171, 178, 180–185, 187–193, 195–199, 201, 203, 204, 214, 223, 225, 226, 235, 249
Light-Emitting Diode, *see* LED
Liquid-Crystal Display, *see* LCD
local memory, 274
local shared bus, 38
Loihi, 263
Lovelace, Ada, 1, 2
low power consumption, 267
low-level libraries, 124
lower precision, 173
LPDDR4, xviii, 266, 267
LSTM, xviii, 162
LTD, xviii, 235
LTP, xviii, 228, 235
LUT, xviii, 274
LVCMOS, 45
M4F, 272
MAC, xviii, xix, 43, 54, 55, 68, 275, 276
Machine Graph, 86, 89, 90, 99–101, 104–106, 122
machine learning, 78, 158, 160, 162, 224, 261, 275, 276
Machine Vertex, 86, 88, 91, 100, 101, 106
Manchester, 3, 53, 132
Manchester Art Gallery, 128, 132
Manchester Baby machine, 4
MAP, 139, 141–143
Markov Chain Monte Carlo, 78
Markov property, 150
massively-parallel computer, 73
MATLAB, 171, 176
Matlab Auditory Periphery, *see* MAP
maximum connection distance, 71
MC, xix, 8, 20, 21, 31–33, 44, 47, 50, 141, 172, 270
membrane capacitance, 181
membrane potential, 103, 106, 119–121, 149, 150, 179, 181–183, 205, 218, 226, 228
membrane resistance, 149
membrane time constant, 181
memory bandwidth, 267
memory controller, 267

Memory Protection Unit, *see* MPU, 48
MEP, 278
mesh, 20
microprocessor, x, 4, 18, 19, 127
Microsoft, 129
MII, xix, 43
minimum energy point, *see* MEP
Minitaur, 176, 178
ML, xix, 162, 272, 275, 276
MNIST, xix, 134, 170–172, 174, 176, 177, 197, 198, 238, 247, 258, 259, 261
Model for Structural Plasticity, 237
modulated plasticity, 233
modulator, 234
monitor, 40, 44
monitor processor, 44
Moore's Law, 264, 265
motion detection, 136, 249, 251
movement, 252
MPSoC, 265
MPU, 48
multi-processor, 265
multicast, *see* MC
multiplicative weight dependence, 222
mutation, 260
mutex, 82
mutual exclusion, 39, 41
nack, 64, 65
natural language processing, 162
Nearest Neighbour, *see* NN
NEAT, xix, 257
Nengo, xix, 130, 131
neocortex, 221
NEST, 103, 197, 200, 201
netlist, 277
network topology, 62
Network-on-Chip, *see* NoC
neural algorithm, 16
neural dynamics, 118
neural networks, 6
neuroevolution, 257
neuroinformatics, 11
neuromodulated STDP, 274
neuromodulator, 231
neuromorphic, 6, 50, 54, 55, 57, 141, 153, 170, 178, 201, 203, 204, 236, 250, 266, 279
neuromorphic applications, 267
neuromorphic computing, 265
neuromorphic hardware, 142, 196
neuromorphic system, 56, 261, 262
NEURON, 103
neuron model, 181
neuron model parameters, 255
neuron update, 111
neuroprosthetics, 153
neurorobotics, 78, 129
neuroscience, 141, 279
neurotransmitter, 136, 137, 146, 184, 189, 210, 231–233
NMDA, xix, 228–230
NN, xix, 31, 33, 34, 38, 48, 82, 83, 270
NoC, xix, 14, 15, 38, 39, 41, 44, 48, 51, 54, 266–268, 270–272, 275, 276
noise, 189, 190
Non-Return-to-Zero, *see* NRZ
NP, xix, 148, 152–155, 158
NP-complete, 153
NRZ, xix, 36, 38
NSP, xix, 178, 180, 181, 189–197, 199–201, 203
NVidia, 263
object recognition, 163
ODE, 273
off-chip sensors, 267
OME, 139, 140
OpenAI, 160, 162
optic nerve, 163
optimal packing, 74

optimisation, 45, 261
optimisation algorithm, 259
optimisation framework, 261
optimisation method, 196
oriented edges, 163
ossicle bones, 139
out-of-credit, 64, 65
outer and middle ear, *see* OME
Outgoing Edge Partition, 101
overhead, 273
P, xx, 148, 153, 154
P2P, xx, 31, 49, 83
Pac-Man, 257
packet received event, 108, 112
packet transit time, 110
packets, 79, 272
PACMAN, 84
PAF, xx, 178, 191, 194–197, 201, 203
parallel I/O, 43
Paris, 171
payload, 44
PC, xx, 15, 129
PCB, xx, 12, 36, 43, 49, 56, 58
PE, xx, 265–268, 270–272, 275, 277–279
Peri Stimulus Time Histogram, 141
periodic boundary conditions, 250
peripheral device, 39
Phase-Locked Loop, *see* PLL
PHY, xx, 43, 267
Physics, 147
PID controller, 130
place and route, 267, 277
plasticity, 127
PLL, xx, 39, 279
Point-to-Point, *see* P2P
Poisson, 103, 104, 122, 145, 150, 152, 170, 172, 180, 184, 187, 192, 195, 197, 198, 214, 222, 225, 226, 228, 234, 246, 248, 249, 251
polynomial time, 148
POST, xx, 60
potentiation, 211
power analysis, 277
power consumption, 18, 36, 40, 44, 59, 60, 67, 99, 146–148, 196, 265, 267
power dissipation, 44, 177
Power Spectral Density, *see* PSD
Power-On Self Test, *see* POST
precision, 252
PRNG, xx, 273
process technology, 50
processor, 18, 30, 38, 39, 44, 48, 49
profiling, 99
prototype, 16, 54, 263, 273
provenance data, 91
PSD, 184
PSP, xx, 150, 179
PSTH, 141, 142
puzzle, 156
PVT, xx, 277–279
PyNN, 84, 103–106, 110, 113, 114, 118, 119, 122, 126, 130, 138, 171, 183, 185, 204, 207, 219, 222, 250, 256
Python, 84, 85, 87, 89, 91, 97, 99, 103, 106, 119, 122, 171, 227
QPE, xxi, 266–268, 270, 279
Qualcomm, 263
quantum computer, 148
RAM, xxi, 8, 10, 14, 19, 25–27, 31, 34, 44, 46, 48, 49, 51, 147
Ramón y Cajal, 3
random initialisation, 259
random mutation, 256
random number generator, 272
rank-order encoding, 250, 254
Raspberry-Pi, 146, 147
RBM, xxi, 161, 162, 170, 189
read-modify-write operation, 39

Read-Only Memory, *see* ROM
readout neurons, 250
real time, 109, 110, 116, 140, 141, 143
Realising Our Potential Award, *see* ROPA
realtime, 124
realtime execution, 116
realtime simulation, 105
recall, 252
receptive field, 133, 135, 161, 173, 174, 237–241, 243–245
redundancy, 62
refractory period, 120, 181, 235
register, 41
Reichardt detector, 136
reinforcement learning, 161, 162, 231
ReLU, xxi, 181, 183, 189, 195–201, 203, 276
request-and-response mechanism, 96
research, 279
reset potential, 120
ResNet, 162
response latency, 202
resting potential, 106, 171
Restricted Boltzmann Machine, *see* RBM
ReSuMe, 232
retina, 133, 163
retinal ganglion cell, 246
RISC, xxi, 4
RK-2 midpoint, 121
RMSE, xxi, 247, 252
RNN, xxi, 162
robot, 80, 129, 132, 203, 279
ROM, xxi, 14, 26, 27, 29, 30, 39, 40, 43, 54, 60, 67, 81
ROPA, 5
routeing, 139
routeing table, 79, 80, 84, 94, 106
router, 8, 21, 23, 26, 29–32, 36, 38, 42, 44, 61, 79, 268–270, 272
Royce, Sir Henry, 52
RTL, 45
RTZ, xxi, 36
run-time modification, 47
Rx, 63, 66
salient stimuli, 143
SARK, xxi, 80, 81, 84, 105, 107
SATA, xxi, 57–60, 66–68, 73, 79
satisfiability, 154
satisfiable, 152
SCAMP, xxi, 80, 82–85, 95, 96
scheduler thread, 108
scheduling of allocations, 74
script, 84
SDP, xxi, 80, 81, 95, 96
SDRAM, xxii, 10–16, 19, 24, 27–30, 38–40, 43–46, 48, 51, 54, 60, 61, 79, 80, 85, 88–96, 100, 101, 106–108, 112–114, 118, 122, 123, 139, 165, 214, 265, 266, 272
search, 152
self-organisation, 250
sensory periphery, 138
SerDes, xxii, 266
Serial AT Attachment, *see* SATA
Siegert, 170, 171, 179–185, 187–189
Sigmoid, 170, 180, 189
Silicon retina, 55
Silistix, 14
simulation, 143
simulation timestep, 110, 111
simulator, 103
small world, 102
SNN, xxii, 77, 102–105, 107, 110, 116, 117, 119, 122, 127, 128, 136, 138, 149–158, 161, 170, 171, 176–181, 188, 189, 191, 192, 195–197, 199–204, 206, 207, 231, 250, 255, 257–262, 264, 268
SNr, xxii, 144, 145
SoC, xxii, 12, 15, 16, 265

Softplus, 183, 195, 197, 199–203
software, 18, 40, 55, 77, 127, 140, 264, 279
software interrupt, 42
solver, 155
Spalloc, xxii, 74, 259
sparse excitatory connections, 250
spDMA, 272
SPI, xxii, 43, 66
spI/O, 66
spike event, 266
spike processing, 109
spike processing pipeline, 117
spike-driven, 165
SpiN1API, xxii, 81, 84, 105, 107, 108
SpineML, 145–147
SpiNN-1, 54
SpiNN-2, 54
SpiNN-3, 54–56, 84
SpiNN-4, 54
SpiNN-5, 54, 55, 57–63, 66–69, 71, 74, 84, 98
SpiNNak-Ear, 127, 137–143
SpiNNaker, xi, xxii, 1, 4, 8–10, 16–20, 24, 27, 30, 31, 36, 38–45, 47, 48, 50–69, 71–80, 82, 84–89, 95–99, 101, 103–108, 110, 116, 118, 120, 122, 127–130, 133, 138–147, 149, 152, 163–172, 176–178, 203, 205, 207, 210, 213–217, 220, 222, 227, 229, 233, 234, 236–238, 241, 248, 250, 251, 256–262, 264–266, 268–270, 272–274, 279
  Application edge, 87
  Application graph, 87
  Application vertex, 87, 88
  Machine edge, 87, 88
  Machine graph, 87
  Machine vertex, 87
  SDP, 80
  SpiN1API, 81
  SpiNNaker boards, 79
  SpiNNaker machine, 78, 84, 85, 88
  SpiNNaker router, 79
SpiNNaker Application Runtime Kernel, *see* SARK
SpiNNaker chip, 79
SpiNNaker Control And Monitor Program, *see* SCAMP
SpiNNaker Datagram Protocol, *see* SpiNNaker, SDP
SpiNNaker machine, 48, 53, 57, 61, 67, 69, 74, 88, 97, 101, 104
SpiNNaker1, 262, 265, 267
SpiNNaker1M, 53, 60, 68, 71–76
SpiNNaker2, 124, 262, 264–270, 272–274, 276, 278, 279
SpiNNer, 72, 73
spiNNlink, xxiii, 63–66, 73, 79
SpiNNTools, xxiii, 77, 78, 86, 88–90, 93, 94, 97, 101, 106, 111
sPyNNaker, xxiii, 77, 103–107, 109, 111, 112, 116–118, 120–122, 146, 250
SRAM, xxiii, 10, 15, 19, 25–27, 29, 30, 39, 40, 46, 48, 270, 272, 275–277
standard deviation, 183
standard error of the mean, 244
Starburst Amacrine Cells, 136
Starcraft 2, 160
static timing analysis, 277
STDP, xxiii, 205–220, 222, 224–227, 231–233, 235–237, 239, 240, 247, 254, 274, 275
stimulus-specific adaptation, 143
stochastic gradient descent, 196
stochastic process, 182
stochastic search, 154
strategic data, 206

Striatum, 144
structural plasticity, 236, 237, 247, 249–251
sub-threshold, 110, 118, 120, 277
sub-threshold dynamics, 120
Sudoku, 155, 156
supercomputer, 69
superior colliculus, 238, 254
supervised learning, 228, 239, 246
supervision, 229
supply voltage, 277
SWTA, xxiii, 227, 228
symbol recognition, 166
synapse event, 107
synaptic contact, 237
synaptic delay, 11, 15, 213–215, 249, 251–253
synaptic efficacy, 179, 207, 208
synaptic event, 279
synaptic input, 118
synaptic input buffer, 110, 111
synaptic matrix, 112–114, 213–215
synaptic plasticity, 207, 249, 250
synaptic rewiring, 236–239, 242, 245–247, 249
synaptic weight, 103, 165, 213, 222, 237, 266
synchronisation, 39, 41, 93
Synchronous Dynamic Random-Access Memory, *see* SDRAM
synthesis, 45, 65, 267, 277
System Controller, 40, 41, 60, 82, 99
t-test, 251–253
TCAM, xxiii, 9, 10, 32, 34, 35, 47, 49, 270
TCM, xxiii, 27, 28, 48
Tensilica, 263
Tera Instructions Per Second, *see* TIPS
Ternary Content-Addressable Memory, *see* TCAM
thalamus, 144
The Imitation Game, 128
threads, 81
threshold, xviii, 106, 118–121, 149, 165, 179, 181, 183, 223, 225, 229, 235, 246, 279
Thumb, 80
Tightly-Coupled Memory, *see* TCM
time constant, 120, 149, 171, 179, 181, 183–185, 187, 188, 194, 195, 208, 210, 211, 217, 218, 223, 274
timer event, 109, 110, 115–117, 122, 123
TIPS, 15
TM, xxiii, 138, 139
topographic map, 238–241, 243, 245
topology, xi, 20, 33, 38, 57, 62, 69–71, 102, 104, 106, 146, 151, 268
torus, 20, 49, 57, 61, 62, 69–71, 269
traffic visualiser, 98
training accuracy, 259
transcendental functions, 274
transistor, xv, xxiv, 44, 51
true random, 273
TrueNorth, 181, 263
Turing test, 4
Turing, Alan, 3, 4, 128
Tx, 63, 65, 66
Tympanic Membrane, *see* TM
UDP, xxiii, 80–82, 97
ultra-low voltage, *see* ULV
ULV, 277
undecidable, 148
uniform distribution, 106, 226, 228, 237
Universite Pierre et Marie Curie, 171
University of Manchester, 9, 53, 74, 256
unlabelled data, 161
unsatisfiable constraints, 157
unseeded population, 259
unsolvable, 148

unsupervised learning, 161, 170, 227, 229, 247, 249, 250
user, 44, 47, 48, 55, 74, 77, 78, 84, 89, 92, 103–105, 107, 108, 110, 112, 116–119, 139, 142, 143, 146, 178, 219, 259, 273, 274, 279
user event, 107, 108, 112, 117
verifiable, 148
Verilog, 45
VGG Net, 162
VIC, xxiii, 49
virtual machine, 74, 85
vision system, 163
visual cortex, 161, 163
visual pattern recognition, 258, 264
visualisation, 98, 103, 238, 248
VLSI, xxiv, 5, 6
VNW, 277
VPW, 277
watchdog, 29, 30, 37, 40–42, 46
weight precision, 172–176
Wiesel, 161
Williams, Freddie, 3
WTA, xxiv, 151, 250, 254
Xilinx, 58, 66
XML, 146
XOR, 257, 273
ybug, 84
Zero-Insertion-Force, *see* ZIF
ZIF, xxiv, 54
Zurich, 170

About the Editors

Steve Furber CBE FRS FREng is ICL Professor of Computer Engineering in the Department of Computer Science at the University of Manchester, UK. After completing a BA in mathematics and a PhD in aerodynamics at the University of Cambridge, UK, he spent the 1980s at Acorn Computers, where he was a principal designer of the BBC Microcomputer and the ARM 32-bit RISC microprocessor. Over 130 billion variants of the ARM processor have since been manufactured, powering much of the world's mobile and embedded computing. He moved to the ICL Chair at Manchester in 1990 where he leads research into asynchronous and low-power systems and, more recently, neural systems engineering, where the SpiNNaker project has delivered a computer incorporating a million ARM processors optimised for brain modelling applications.

Petruț Bogdan is a Research Associate in the Department of Computer Science at the University of Manchester, UK. He received a BSc in Computer Science in 2016 and a PhD in Computer Science in 2019, both from the University of Manchester. His research focused on structural plasticity and real-time modelling of spiking neural networks on SpiNNaker.

Contributing Authors

Petruț Bogdan
The University of Manchester

Christian Brenninkmeijer
The University of Manchester

Dave Clark
The University of Manchester

Simon Davidson
The University of Manchester

Andreas Dixius
TU Dresden

Donal Fellows
The University of Manchester

Gabriel Fonseca Guerra
The University of Manchester

Steve Furber
The University of Manchester

Andrew Gait
The University of Manchester

Francesco Galluppi
Gensight Biologics

Jim Garside
The University of Manchester

Jonathan Heathcote
BBC Research and Development

Sebastian Höppner
TU Dresden

Michael Hopkins
The University of Manchester

Dongwei Hu
The University of Manchester

Robert James
The University of Manchester

Edward Jones
The University of Manchester

Florian Kelber
TU Dresden

James Knight
The University of Sussex

David R. Lester
The University of Manchester

Qian Liu
aiCTX, Zurich

Gengting Liu
The University of Manchester

Christian Mayr
TU Dresden

Mantas Mikaitis
The University of Manchester

Felix Neumärker
TU Dresden

Johannes Partzsch
TU Dresden

Jeffrey Pepper
The University of Manchester

Adam Perrett
The University of Manchester

Garibaldi Pineda García
The University of Sussex

Luis A. Plana
The University of Manchester

Oliver Rhodes
The University of Manchester

Andrew G. D. Rowley
The University of Manchester

Stefan Schiefer
TU Dresden

Stefan Scholze
TU Dresden

Basabdatta Sen-Bhattacharya
BITS Pilani, Goa Campus

Teresa Serrano Gotarredona
Instituto de Microelectrónica de Sevilla

Delong Shang
The University of Manchester

Alan B. Stokes
The University of Manchester

Marco Stolba
TU Dresden

Evangelos Stromatias
Medis Medical Imaging Systems

Steve Temple
MindTrace