LEARNING IN SPIKING NEURAL NETWORKS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2012

By
Sergio Davies
School of Computer Science

Contents

Word Count: 37306

List of Tables

List of Figures

List of Algorithms

Abstract

LEARNING IN SPIKING NEURAL NETWORKS

Sergio Davies
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2012

Artificial neural network simulators are a research field which attracts the interest of researchers from various fields, from biology to computer science. The final objectives are the understanding of the mechanisms underlying the human brain, how to reproduce them in an artificial environment, and how drugs interact with them. Multiple neural models have been proposed, each with their peculiarities, from the very complex and biologically realistic Hodgkin-Huxley neuron model to the very simple "leaky integrate-and-fire" neuron. However, despite numerous attempts to understand the learning behaviour of the synapses, few models have been proposed. Spike-Timing-Dependent Plasticity (STDP) is one of the most relevant and biologically plausible models, and some variants (such as the triplet-based STDP rule) have been proposed to accommodate all biological observations. The research presented in this thesis focuses on a novel learning rule, based on the spike-pair STDP algorithm, which provides a statistical approach with the advantage of being less computationally expensive than the standard STDP rule, and is therefore suitable for its implementation on stand-alone computational units. The environment in which this research work has been carried out is the SpiNNaker project, which aims to provide a massively parallel computational substrate for neural simulation. To support such research, two other topics have been addressed: the first is a way to inject spikes into the SpiNNaker system through a non-real-time channel such as the Ethernet link, synchronising with the timing of the SpiNNaker system. The second research topic is focused on a way to route spikes in the SpiNNaker system based on populations of neurons. The three topics are presented in sequence after a brief introduction to the SpiNNaker project. Future work could include structural plasticity (also known as synaptic rewiring); here, during the simulation of neural networks on the SpiNNaker system, axons, dendrites and synapses may be grown or pruned according to biological observations.

Declaration

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

Copyright

To Nathaële, for her support and patience.
To William and Maria Rosaria, for their help and encouragement.

Acknowledgements

I wish to thank all those who helped me getting to the University of Manchester: my parents, Nathaële, and all the friends who encouraged and supported me in following my idea (sometime, giving me a kick and encouraging me to move on). The names are too many to list here, so that I realize right now how many friends I can count on. A particular acknowledgement goes to Leslie who helped me with my English, when, at the beginning of this adventure, it was far from being reasonable!

And a big thank you goes to all those people that I met during my stay in Manchester and that have been with me during these three years, who celebrated with me achievements and who helped me to relax when I was too stressed.

In the same way, I cannot forget all the friends that from Italy have always continued to give me their support even at $3,000Km$ distance!

Secondly, but not in importance, a big thank you to everyone at the University of Manchester who helped me in completing this course.

I would like to thank my supervisor, Professor Steve Furber, for his precious guide during this Ph.D. course.

Also a kind "Thank you!" goes to Dr. Jim Garside, my co-supervisor, a very helpful guide who has always been free for long discussions.

Special thanks go to Dr. John Viv Woods, for all the cooperation in preparing this thesis and for patiently proof-reading all the papers that I have written for publication. In particular I admire that he has always a smile on his face and a joke to say to keep up the spirit.

A kind thank you goes to all those within the SpiNNaker group (and broader within the APT group) who supported me during these three years, introducing me to the neural network community. Some of them became friends even out of the lab, with a beer or a badminton racket in the hand! In particular I would like to thank Alexander Rast, Cameron Patterson, Javier Navaridas, Luis Plana and Thomas Sharp (alphabetically sorted) for making these three years always interesting.

Acknowledgements

I am grateful to the Engineering and Physical Science Research Council (EPSRC) which funded my Ph.D. course.

A "Thank you!" also goes to Dr. David Lester, who introduced me to wine tasting. Some days it became hard to work after those kind of practical lessons!

Another special "Thank you!" goes again to Professor Steve Furber for funding my travels around the globe for research and presentations:

1. 25th April 2010 - 8th May 2010 — CapoCaccia cognitive neuromorphic engineering workshop — Capocaccia, Sardinia, Italy;
2. 27th June 2010 - 14th July 2010 — Telluride neuromorphic cognition engineering workshop — Telluride, Colorado, U.S.A.;
3. 18th July 2010 - 23rd July 2010 — International Joint Conference on Neural Networks (IJCNN) 2010 held within the World Congress on Computational Intelligence (WCCI) 2010 — Barcelona, Spain;
4. 22nd November 2010 - 25 November 2010 — International Conference On Neural Information Processing (ICONIP) 2010 — Sydney, Australia;
5. 3rd May 2011 - 5th May 2011 — ACM Computing Frontiers conference — Ischia, Campania, Italy;
6. 6th May 2011 - 14th May 2011 — CapoCaccia cognitive neuromorphic engineering workshop — Capocaccia, Sardinia, Italy;
7. 26th June 2011 - 16th July 2011 — Telluride neuromorphic cognition engineering workshop — Telluride, Colorado, U.S.A.;
8. 18th July 2011 — Invited talk at University of California S. Diego — S.Diego, California, U.S.A.;
9. 19th July 2011 — Invited talk at Qualcomm and Brain Corporation — S.Diego, California, U.S.A.;
10. 20th July 2011 — Invited talk at Salk institute — S.Diego, California, U.S.A.;
11. 28th July 2011 — Invited talk at University of Stanford — California, U.S.A.;
12. 31st July 2011 - 5th August 2011 — International Joint Conference on Neural Networks (IJCNN) 2011 — San Jose, California, U.S.A.;

13. 12th September 2011 - 13th September 2011 — FACETS-ITN neuromorphic systems workshop — Heidelberg, Germany;

14. 29th April 2012 - 13th May 2012 — CapoCaccia cognitive neuromorphic engineering workshop — Capocaccia, Sardinia, Italy;

15. 10th June 2012 - 15th June 2012 — International Joint Conference on Neural Networks (IJCNN) 2012 held within the World Congress on Computational Intelligence (WCCI) 2012 — Brisbane, Australia;

16. 1st July 2012 - 21st July 2012 — Telluride neuromorphic cognition engineering workshop — Telluride, Colorado, U.S.A.

Total: about $150,000Km$; ≈ 3.75 times the earth circumference, in ≈ 4.5 months! Considering the term of three years for my Ph.D., I have been travelling at $\approx 6Km/h$ for the whole period, writing 1 word of this thesis every $\approx 4.0Km$!

The Author

Sergio Davies was born in 1981 in Naples, Italy. He graduated in 2006 from the University of Naples "Federico II" in telecommunication engineering with a specialization in computer science.

During his studies he met Prof. G. Ventre and Prof. G. Iannello who introduced him in the research world benchmarking parallel architectures in the Computer Science Department (DIS) at the University of Naples.

After graduation he started a Ph.D. course at the University of Salerno on pattern recognition applied to high speed motorways, but a lack of funds forced him to quit it.

He then entered industry in 2007 working for Nolan, Norton Italia - KPMG. He provided IT strategic advice to Banks, Ministries, public institutions and big telecommunication companies.

In 2009 he applied for a Ph.D. position at the University of Manchester, where he started the course in late September working in the SpiNNaker group supervised by Prof. Steve Furber.

His main research interest is synaptic plasticity in spiking neural networks. He is also interested in self-reconfiguring architectures for spiking neural networks simulation, with the idea of experimenting synaptic rewiring models in the SpiNNaker system.

Chapter 1

Introduction to neural networks

1.1 Neural networks

The term "neural network" is used to refer to a circuit of neurons that performs information processing using an approach to the problem different from the usual algorithmic computation of modern computer systems. The basic computational units of neural networks are neurons which may be either biological, or simulated mathematically on a computational substrate (e.g. a standard computer, an analogue circuit, etc).

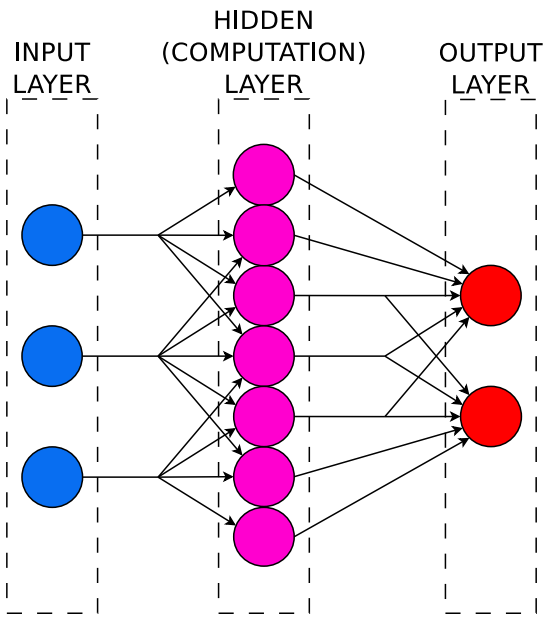


Figure 1.1: General structure of a simple feed-forward artificial neural network. Each coloured disc is a neuron, and each arrow is a synapse.

A simple example of a feed-forward neural network can be expressed in the three-layer network depicted in Fig.1.1, where each layer performs a simple task and the whole network performs an algorithm.

The group of neurons on the left is the input layer whose task is to collect stimuli from the external world; these neurons behave in a way similar to the sensory neurons in humans. The middle layer is the computational layer, which processes information received by the input layer (or perceived by senses in life forms); these neurons can be compared with the brain in humans. Finally, the output layer, on the right, generates the output stimuli of the network; these neurons are similar to the motor neurons in humans.

More complex structures of neural networks may include recurrent connections (connections between neurons of the same layer) or backward propagation: a feedback channel from the computation and/or the output layer to the previous layer(s). Further examples of neural network architectures exist in the literature.

This short and simplified example did not define any characterization of the neurons and synapses in use. However, on the basis of the characteristics of these components, the class of neural networks, and the type of algorithm that they perform, can be defined. The two main classes into which neural networks are divided are defined by the type of neuron in use: biological neurons (therefore biological neural networks) and artificial neurons (therefore artificial neural networks).

In this first chapter the biological aspects of neural networks are examined so that it is possible to compare them with the models used in artificial neural networks. After the comparison there is a description of the objective of this research work with the contributions and the structure of this thesis.

1.1.1 Biological neural networks

Biological neural networks are those formed by biological neurons which are connected to carry out the functionalities typical of the nervous system in biological life forms.

However, not all multicellular life forms on the earth have a nervous system. For example, sponges are very old life forms comprising colonies of cells which do not have a nervous system (Sanes et al., 2000) to allow electrical communication between the various part of the body. In particular the presence and the complexity of the nervous system has evolved during the various historical eras.

In the evolutionary scale, with regard to the nervous system, after the sponges it

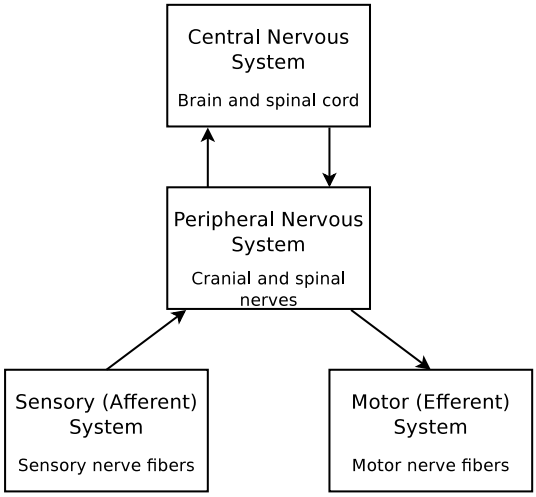


Figure 1.2: Simplified structure of the nervous system in vertebrate animals.

is possible to find the "Radiata" branch: life forms which have a radial symmetry. In these life forms it is possible to identify a "top" and a "bottom", but not "left" and "right" sides; jellyfish are an example of this class of animal. These life forms have a simple nerve net which allows reactions to external stimuli and in some cases even complex behavioural patterns (Garm et al., 2007).

Life forms having the most complex neural system belong to the "Bilateria" branch. These are the life forms which are (approximately) symmetric with respect to a longitudinal axis, and for which it is possible to define a "left" and a "right" side in addition to a "top" and a "bottom". This branch includes the human race which has the most complex nervous system known in nature: ≈ 100 billion neurons (average) (Williams and Herrup, 1988) and 0.15 quadrillion ($1.5 * 10^{14}$) synapses in just the neocortex (Pakkenberg et al., 2003).

More generally, in the "Bilateria" branch, the "Vertebrate" subphylum occurs. In individuals of this subphylum, the nervous system can be divided into two interconnected halves (Kandel et al., 2000): the peripheral nervous system and the central nervous system. However, these two parts differ only from an anatomical point of view (see Fig.1.2). From a functional point of view there is a strict relationship between the peripheral and the central nervous system. The peripheral nervous system is the Input/Output interface for the central nervous system: stimuli collected through the senses are sent through the peripheral nervous system, as sequences or trains of spikes, to the central nervous system, where they are processed (by means of spikes, with the help of neuromodulators). The central nervous system is the main processing

unit for this information where all physical and behavioural reactions are processed and subsequently stimuli are sent back through the peripheral nervous system to the motor system generating the physical reaction.

1.1.2 Artificial neural networks

Artificial neural networks are circuits made by the interconnection of artificial neurons, which mimic the behaviour of biological neurons. This type of neural network needs to be simulated through a custom component designed for this purpose. Simulators comprise hardware (analogue or digital) or software (digital) components which compute mathematical models of biological neurons and biological synapses.

If neural networks are classified according to computational units (i.e. neurons and synapses), it is possible to distinguish three classes of algorithms (Maass, 1997) which differ between them by the types of input/output signals they are able to process and to emit: binary signals, continuous values and spike events (see Fig.1.3). This research work focuses on the third class of artificial neural networks: networks with spiking neurons; however, for completeness, here we describe also the other two classes of artificial neurons.

The first class of artificial neural networks features neurons, also called perceptrons, which are composed each of two sections: a sum and a threshold. The sum part receives input from a set of weighted synapses (in Fig.1.3 $w1,\ldots,wN$ represent the set of input weights) and performs a thresholding function on the result of the sum. Both the input and the output have values that may be equal to either 0 or 1 (discrete values).

The second class of artificial neural networks is composed by neurons which perform a two-stage computation: a sum of values received through weighted synapses, and a sigmoid function evaluator whose input is the result of the sum previously computed. The second computational stage gives the name to the neurons, which are also called sigmoidal units. The inputs can be any real-valued number, and the output is defined by the transfer function – the sigmoid unit for example limits outputs to $[0;1]$, whereas the hyperbolic function produces outputs in the range $[-1;1]$.

The third class of neural network is composed by spiking neurons: neurons which communicate through short signals, called spikes. This class of neural network has two main differences when compared with the previous two classes. In the first place, this class of neurons introduce the concept of time in the simulation, while earlier the neural networks were based on abstract steps of simulation. In addition, such neurons present similarities to biological neurons, as they both communicate using short signals, which

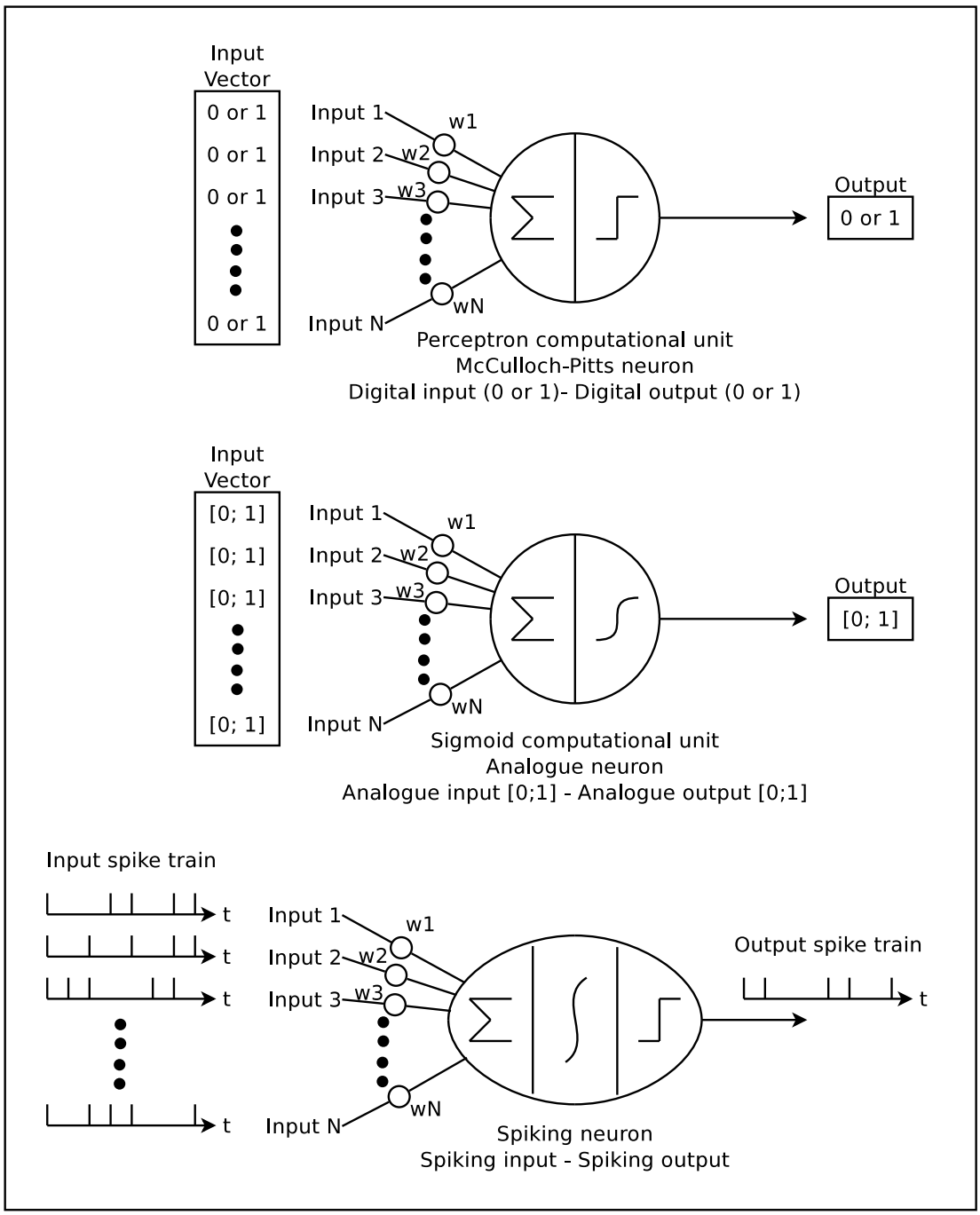


Figure 1.3: Classes of artificial neurons.

in biology are electric pulses (spikes), also known as action potentials.

In biology, the axon of a neuron is connected to the dendrites of another neuron through synapses. Such structures have the ability to "pump" electrical charges from the intracellular medium into the post-synaptic neuron. The amount of charge is a

characteristic of the specific synapse and the specific neurotransmitter in use.

In artificial neural networks, to develop a mathematical model that reproduces such a process, various parameters are used (synaptic weight, synaptic delay, synaptic current function, etc.), to represent the characteristics of each synapse. For the synaptic plasticity discussed in this work, we focus in particular on the synaptic weight parameter; however, other parameters are also presented for completeness. In particular, in Fig.1.3, the set $w1, \ldots, wN$ represents the input synaptic weight for each of the synapses.

Each spiking neuron of this class is composed of three computational stages: a sum for all of the neuron's input current, which is then integrated over time (the second stage of the computation), and then, when the neuron's membrane potential (the potential difference between the two sides of the cellular membrane) raises above a threshold, in the third stage of the computation, a spike is emitted and the membrane potential returns to a reset value. The mathematical model of such spiking neurons is generally described through Ordinary Differential Equations (ODE). Sections 3.6.1 and 3.6.2 present examples of such models.

Table 1.1, proposed by Dr. E. Izhikevich (Izhikevich, 2004), compares several neuron models in terms of biophysical meaning, the types of biological neurons that the model is able to replicate, the ability to exhibit autonomous chaotic behaviour and the number of (floating-point) operations required for each step of simulation (with the hypothesis of a $1msec$ simulation step). A "+" in the table means that the model is able to emulate the particular type of behaviour. A "-" means that the model isn't able to reproduce it. A white space means that theoretically the model allows the specific behaviour, but it was not possible to find the appropriate parameters (in a reasonable amount of time).

Neuron Models	Number of FLOPS	5	10	13	10	7	13	72	120	600	180	1200
Integrate-and-Fire	Chaos	-	-		+	-	+	-	+	-		+
Integrate-and-Fire with adaptation	Inhibition-induced bursting	-	-	-	-	-	+	-		-		
Integrate-and-Fire-or-Burst	Inhibition-induced spiking	-	-	-	-	-	+	+	+	+		+
Resonate-and-Fire	Accommodation	-	-	-	+	-	+	+	+	+	+	+
Quadratic Integrate-and-Fire	Depolarizing after-potential	-	+	+	+	-	+	-	+	-	+	+
Izhikevich	Bistability	-	-	+	+	+	+	+	+	+		+
FitzHugh-Nagumo	Threshold variability	-	-	-	-	+	+	+	+	+	+	+
Hindmarsh-Rose	Rebound burst	-	-	+	-	-	+	-	+		+	+
Morris-Lecar	Rebound spike	-	-	+	+	-	+	+	+	+	+	+
Wilson	Integrator	+	+	+	+	+	+	-	+	+	+	+
Hodgkin-Huxley	Resonator	-	-	-	+	-	+	+	+	+	+	+
	Subthreshold oscillations	-	-	-	+	-	+	+	+	+	+	+
	Spike latency	-	-	-	-	+	+	+	+	+	+	+
	Class 2 excitable - spike latency	-	-	-	+	-	+	-	+	+	+	+
	Class 1 excitable	+	+	+	+	+	+	+	+	+	+	+
	Spike Frequency adaptation	-	+	+	-	-	+	-	+	-	+	+
	Mixed mode	-	-	-	-	-	+	-		-		
	Phasic bursting	-	-	+	-	-	+					
	Tonic Bursting	-	-		-	-	+	-	+	-	+	+
	Phasic spiking	-	-	+	+	-	+	+	+	+	+	+
	Tonic spiking	+	+	+	+	+	+	+	+	+	+	+
	Biophysically meaningful	-	-	-	-	-	-	-	-	+	-	+

Table 1.1: Comparison between spiking neuron models (Izhikevich, 2004).

1.2 Uses of neural networks

Research on neural networks has generated increasing interest over the last few years. Research groups applied them successfully to various tasks, such as (Misra and Saha, 2010):

- Pattern and sequence recognition;
- Data processing (filtering, clustering and compression);
- System identification and control;
- Game playing and decision making;
- Medical diagnoses;
- High energy physics;
- Image/object recognition;
- Image segmentation;
- Generic image/video processing;
- Finger print feature extraction;
- Autonomous robotics;
- Optical character/handwriting recognition;
- Acoustic sound recognition;
- Real-time embedded control;
- Security control units (e.g. in airports) – use automatic face recognition to raise security levels (Zhang and Fulcher, 1996);
- Control theory with neural networks – the Neural Engineering Framework (Eliasmith and Anderson, 2004)
- ... and much more...

As artificial neural networks get closer to biological examples, it becomes possible to emulate parts of the biological nervous system to study processes which normally happen in the brain, but are not yet completely understood. In particular, spiking neural network simulators allow the study of the extent to which theory can approximate reality in the study of processes such as:

- Verification of how theoretic models approximate biological processes;
- The study of mental illnesses by emulating brain disorders;
- Testing how drugs affect the brain;

The final scope of research involving spiking neural networks is to understand how the human brain works. Since the beginning of computer science, when the first computer executed its program, processors have always consumed high levels of electrical power compared with biological neural networks (e.g. the human brain (Furber, 2008)), but are not even near to being able to mimic what nature created: the brain. This is an aggregate of neural cells which is very power efficient, where each element does a very small computation, but this computation is extremely fast.

The power of the brain is believed to stem from the massive parallelism of interconnected elements, and this is a source of inspiration for novel projects. Scientists try to emulate these characteristics on the basis that these may represent the future of computer technology.

1.3 Adaptability (plasticity) in neural networks

Biological neural networks have the remarkable ability to be able to adapt to the surrounding environment, and to improve their performance (according to a specific measure) in accomplishing specific tasks. Artificial neural networks have been designed to replicate such behaviour, with some degree of accuracy, even though the complete process is not yet fully understood.

The learning ability in biological neural networks is shown in the modification of the parameters of a network (e.g. synaptic weight modification) so that it preserves the experience gained through the learning process.

However learning cannot be defined uniquely, as there are too many activities involved, and this process may be seen from different points of view. For example, the

meaning of learning in an educational environment is quite different from the meaning of the same word in a psychological sense or in a biological environment.

For the purpose of this thesis, the definition of learning in neural networks given by S. Haykin (Haykin, 1999) is most suitable:

> *"Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place"*

In this definition there is a process involved, where these steps need to take place:

1. The environment in which the network resides gives stimuli to the network;
2. The free parameters of the network are modified according to the incoming stimuli;
3. The network generates a new output corresponding to the incoming stimuli from the environment, where modifications occur because of the changes described in the previous step.

The second of these steps is fully described through a *learning algorithm*: a well-defined set of rules which describe how the modifications in the network occur as a consequence of the incoming stimuli.

After modifications take place, the network generates an output, and therefore interacts with the environment. The *learning paradigms* depend on the type of environment which surrounds and provides training signals to the neural network.

1.3.1 Plasticity in biological neural networks

In biology the connection between two neurons shows a high degree of plasticity in the parameters which characterize it. The weight of the synapse is one (and perhaps the most studied) of these parameters. Other parameters which can be influenced concern the speed of propagation of signals between the two neurons (Bakkum et al., 2008), the generation of new connections (Sanes et al., 2000), the cancellation of existing connections which have a very low influence on the post-synaptic neuron (Sanes et al., 2000), and others.

This thesis focuses, in particular, on synaptic weight modification, because this is the most studied in biology and therefore provides the largest amount of supporting material to be emulated using an artificial neural network simulator.

The first hypothesis for the underlying mechanism of the synaptic weight modification was proposed by Hebb in 1949 (Hebb, 1949):

> *"When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased"*
> Hebb, D.O. (1949), "The organization of behavior", New York: Wiley, p. 62

> *"When one cell repeatedly assists in firing another, the axon of the first cell develops synaptic knobs (or enlarges them if they already exist) in contact with the soma of the second cell."*
> Hebb, D.O. (1949), "The organization of behavior", New York: Wiley, p. 63

> *"The general idea is an old one, that any two cells or systems of cells that are repeatedly active at the same time will tend to become 'associated', so that activity in one facilitates activity in the other."*
> Hebb, D.O. (1949), "The organization of behavior", New York: Wiley, p. 70

This mechanism has often been summarized as "cells that fire together, wire together". However, more biologically detailed mechanisms have been described in the last 30 years. Synaptic weights have shown the ability to be either enhanced (potentiated) or depressed, and the time range over which these modifications lasts can span from few milliseconds to a lifetime (Cowan et al., 2001).

These modifications have been associated with adaptation to sensory input and short-term memory as well as long-term memory (collection of experiences), and others.

It is possible to identify two main forms of synaptic plasticity connected with learning and memory which act on the synaptic weights where the difference is the time scale: short-term and long-term plasticity.

In addition to these, there is one further form of adaptation of neural networks which spans a time period which is longer than the life of an individual: evolution (Nolfi et al., 1994).

Here a small set of the biological processes involved in short-term and long-term plasticity is presented. Detailed description of the biological events is left to more appropriate sources. It is necessary only to be aware that these processes do take place in biological neural networks.

Short-term plasticity

This type of synaptic plasticity leads to modifications in synaptic weights which disappear in a short time (in the range between tens of milliseconds and minutes (Cowan et al., 2001)). The modifications involve either enhancement or depression of the synaptic signal, respectively named short-term enhancement and short-term depression.

The weight modification may be a consequence of the signals which pass through the synapse (homeosynaptic plasticity) or may depend on a third neuron, different from the pre-synaptic and post-synaptic neurons, which acts on the synapse (heterosynaptic plasticity).

Homeosynaptic plasticity

This modification acts on the pre-synaptic transmitter release from the synaptic knob either facilitating or inhibiting the release of the neurotransmitters. The general rule (Zucker and Regehr, 2002) describes this learning rule on the basis of the inter-spike time carried by the synapse: if this is less than $20msec$ the synapse is depressed (Paired Pulse Depression - PPD). On the other side, if the inter-spike time is in the range $20-500msec$, then the synapse is potentiated (Paired Pulse Facilitation - PPF) (Cowan et al., 2001).

Heterosynaptic plasticity

This modification acts on the pre-synaptic transmitter release and is regulated by signals extrinsic to the synapse on which it acts. The modifications induced may be either inhibition or facilitation. In biology there are many examples of heterosynaptic inhibition processes, but fewer examples are available for the facilitation process (Cowan et al., 2001).

Long-term plasticity

Plasticity forms that act for longer times are also present in the brain. Generally it is possible to divide them into two major categories: NMDA (N-Methyl-D-Aspartic acid) receptor-based plasticity and NMDA receptor-independent plasticity (Cowan et al., 2001). While, for the first type of plasticity, there is still some discussion on the expression mechanism, for NMDA receptor-independent plasticity, scientists have agreed that this mechanism occurs at the pre-synaptic end in the hippocampus, cerebellum and corticothalamic synapses.

NMDA receptor-independent plasticity has been widely studied and a learning algorithm, called Spike Timing Dependent Plasticity (STDP), has been proposed as a result of the effects generated in the biological experiments. The suggested rule was first proposed by Bi and Poo in 1998 (Bi and Poo, 1998) and described the biological mechanism, from a phenomenological point of view, without detailed analysis of all the biological elements involved. This algorithm has subsequently been reviewed and improved by others (Abarbanel et al., 2002).

The effects studied in the experiments suggested that the precise timing of the incoming and outgoing spikes may be the cause for the Long Term Depression (LTD) and Long Term Potentiation (LTP) of the synapse. This will be discussed later in the thesis in chapter 6 (as background of the learning algorithm that is proposed).

1.3.2 Plasticity in artificial neural networks

In artificial neural networks there are various aspects of the learning process. It is important to define the environment in which the neural network is trained, this is called the "learning paradigm"; the "learning rule" defines the algorithm through which the neural network adapts to the incoming stimuli and there are several possible tasks that a neural network may be trained to do. These three parameters (learning paradigms, learning rules and learning tasks) are described in the next paragraphs (Haykin, 1999).

Learning paradigms

There are four main paradigms which describe the learning environment in which the neural network operates:

1. **Supervised learning**: In artificial neural networks this learning paradigm is characterized by the presence of sets of inputs and knowledge about the desired

outputs. Having knowledge of both inputs and outputs, the network is able to configure itself to apply the transfer function over inputs not previously seen. In this case the network benefits from the presence of a teacher (the desired output) through an error function which induces a modification in the parameters of the internal structure of the network.

An example of learning rule for neural networks composed by sigmoidal units, which uses this learning paradigm, is error-backpropagation, used in Multi-Layer Perceptrons. The biological realism of such an algorithm (described with pseudo-code in Algorithm 1) has been disputed (Stork, 1989), but some implementations claim to be biologically plausible (e.g. van Ooyena and Roelfsemab, 2003).

The supervised learning paradigm has also been applied to spiking neural networks using a modified version of STDP as the learning rule (Strain et al., 2006). The STDP learning rule will be discussed later in chapter 6.

2. **Unsupervised learning**: This learning paradigm is characterized by the presence of only input training patterns, for which no output is provided. Thus the network must learn to adapt through the "experience" collected from the injected patterns. Once the network has adapted to the statistical regularities of the injected input, it forms an internal representation of the features of the input, and is therefore able to generate new classes based on this representation. This learning paradigm will be used later in this thesis to perform tests for learning algorithms. An example rule of this learning paradigm is the STDP algorithm (Bi and Poo, 1998), which is presented and discussed in-depth in chapter 6. In particular, this algorithm has been extracted after the study of biological phenomena, and is therefore biologically plausible;

3. **Reinforcement learning**: In nature, animals learn how to behave not only by observing specimens of the same race or through a "teacher". Often they experiment with new behaviours in a trial and error process where a reward is present. The right behaviour receives a positive reward and therefore is reinforced; actions that did not produce the desired outcome are penalized. The whole process, here described, has been called "reinforcement learning" in connection with the reward function that reinforces the correct behaviour. A link between the theory behind this type of learning and its biological implementation in neural networks has been found only recently (Ponulak and Kasinski, 2011).

Algorithm Name: ERROR BACKPROPAGATION
Description: This algorithm is the basic method to propagate the synaptic weight modification in an MLP network which was initialized with random synaptic weights. In this example a 3-layer network is considered;

---

apply_input(input);
output = *compute_output()*;
compute_error(output, desired_output);
weights_variation_output_layer =
compute_synaptic_weights_update(weights_output_layer, output);
weights_variation_hidden_layer =
back_propagate_synaptic_weights_modification(weights_variation_output_layer, weights_hidden_layer);
update_synaptic_weights();

Function description:
apply_input(): applies the specific input to the Multi-Layer Perceptron;
compute_output(): computes the output for the Multi-Layer Perceptron;
compute_error(): computes the new synaptic weights as a function of the error between the actual output and the desired output;
compute_synaptic_weights_update(): computes the synaptic weight update for the synapses from the hidden layer to the output layer, given the current synaptic weights and the error value;
back_propagate_synaptic_weights_modification(): computes the synaptic weight update for the synapses from the input layer to the hidden layer, given the weight update for the synapses from the hidden layer to the output layer;
update_synaptic_weights(): applies the synaptic weights update appropriately;

Algorithm 1: Error backpropagation

An example rule of such a learning paradigm applied to neural networks composed by sigmoidal units is the Q-Learning rule (Watkins and Dayan, 1992), which uses temporal differences to estimate the expected reward ($Q(s,a)$) of doing a particular action (a) while in a particular state (s). This learning rule is described by the formula (1.1):

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(s_t,a_t) \times (R_{t+1} + \gamma \times \max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \tag{1.1}$$

where:

- $\alpha(s_t, a_t)$ is the learning rate for the specific action and state ($0 < \alpha \leq 1$);
- R_{t+1} is the reward observed after performing action a_t;
- γ is a discount factor ($0 \leq \gamma < 1$);

This learning paradigm has been applied also in spiking neural networks using STDP as learning rule with a modulatory signal (Farries and Fairhall, 2007).

4. **Evolutionary learning**: Biological evolution is the process through which individuals are selected as the ones that have best adapted to the environment:

   > *"[...] any being, if it vary however slightly in any manner profitable to itself, under the complex and sometimes varying conditions of life, will have a better chance of surviving, and thus be NATURALLY SELECTED."* Darwin (1859), "On the origin of species" (6th edition), New York: P.F. Collier, p. 15.

   This happens through adaptation to the environment on the time span of generations: life forms that have achieved a better adaptation to the environment are also able to live longer and reproduce more. Therefore, through natural selection, the "winning" genes are the most diffused across the individuals of a particular population. As evolution has been described in biology, similar processes can be also described in numerous other fields (Holland, 1975). In particular it has been described in artificial neural networks through evolutionary algorithms. In the experiments ran by Pavlidis et al. (2005), multiple copies of the same structure of the neural network are reproduced on a computational system. Each of these copies is able to learn independently from the others. The network evolves bringing from one generation to the subsequent only the populations of neurons that perform their task in the most effective way.

Learning rules

Learning algorithms for neural networks may be based on a series of principles. Here we cite some of them, leaving detailed explaination to more appropriate sources (e.g. Haykin, 1999):

- **Error-correction learning**: The output generated by the network is compared with (i.e. subtracted from) the requested output, and this error signal is used to make the appropriate changes to the synaptic weights. This rule is used in a supervised learning environment.
- **Memory based learning**: All the injected inputs are compared with those previously stored in memory. The input is then attributed to the class of its closest neighbour that is present in memory.
- **Hebbian learning**: The basis of this learning algorithm was described at the beginning of section 1.3.1 with the quotation of Hebb's postulate. The STDP learning rule belongs to this category.
- **Competitive learning**: The output neurons of a network compete among themselves to become active, inhibiting the others. This rule is used later in this thesis in addition to the Hebbian learning rule described earlier.

Learning tasks

There is a long list of possible tasks to test the learning ability of neural networks (Haykin, 1999). The research in this thesis focuses on a pattern recognition task which includes a learning phase using a set of injected spikes: the network has to extract and memorize the statistical properties of the injected spike pattern and determine if it belongs to a known pattern. This is discussed later in chapter 6.

1.4 Statement of the problem and objectives

Implementing learning (in particular the STDP algorithm, as described later in section 6.3) on an event-based architecture, such as the SpiNNaker system, is computationally very complex, and very expensive from a memory occupation point of view; this is described in section 6.12.1.

The aim of this research is to provide a novel learning algorithm for a neuromorphic system, requires less computational power and occupies less memory compared with other learning rules currently studied, but provides learning features comparable to other algorithms.

1.5 Contributions

In this thesis three main contributions will be presented:

1. A novel implementation of a learning rule suitable for neuromorphic implementation; the rule presented will be called STDP-TTS (Spike Timing Dependent Plasticity with Time-To-Spike forecast).

2. A novel approach to multicast packet routing in the SpiNNaker system; throughout this thesis this topic will be referred to as "population-based routing";

3. A real-time host computer interface to inject spikes into the SpiNNaker system. These may be generated (for example) by a capturing interface (such as a silicon retina) and injected into the system while the simulation is running;

The primary contribution of this work is the learning rule. The other two contributions were necessary to develop a suitable environment to perform the required tests to study the characterization of the novel learning rule. In particular, while the motivation for the novel learning rule was anticipated earlier as it represents part of the research problem, the specific motivations for each of the other two contributions are described in the sections 4.1 and 5.1.

1.6 Publications

1.6.1 Journal articles

- Sergio Davies, Cameron Patterson, Francesco Galluppi, Alexander Rast, David Lester and Steve Furber
  "*Interfacing Real-Time Spiking I/O with the SpiNNaker neuromimetic architecture*"
  published in Australian Journal of Intelligent Information Processing Systems (AJIIPS) 2010, volume 11, number 1, pages 7–11 (Davies et al., 2010); This article represents the basis of the work described in chapter 4;

- Sergio Davies, Alexander Rast, Francesco Galluppi and Steve Furber
  "*A forecast-based STDP rule suitable for neuromorphic implementation*"
  published in Neural Networks – Special Issue 2012 – Selected papers from IJCNN 2012 – August 2012, Volume 32, pages 3–14.
  This article covers the topic presented in the first half of the chapter 6;

The author of this thesis has also contributed to the development and writing of the article listed below. However, this will not be included directly in this work.

- Alexander Rast, Francesco Galluppi, Sergio Davies, Luis Plana, Cameron Patterson, Thomas Sharp, David Lester and Steve Furber
  "*Concurrent Heterogeneous Neural Model Simulation on Real-Time Neuromimetic Hardware*"
  published in Neural Networks, Volume 24, Issue 9, Nov 2011, pages 961–978 (Rast et al., 2011a).

- Xin Jin, Mikel Lujan, Luis A. Plana, Sergio Davies, Steve Temple and Steve Furber
  "*Modelling Spiking Neural Networks on SpiNNaker*"
  published in Computing in Science & Engineering, 2010, volume 12, number 5, pages 91–97 (Jin et al., 2010b);

1.6.2 Conference papers

- Sergio Davies, Alexander Rast, Francesco Galluppi and Steve Furber
  "*Maintaining Real-Time Synchrony on SpiNNaker*"
  published in the proceedings of the 8th ACM International Conference on Computing Frontiers 2011, pages 15:1–15:2 (Davies et al., 2011b).
  This paper describes the solution to a technical issue faced while testing the STDP TTS learning algorithm. The content of the paper is not discussed directly in this thesis, however the work presented allowed the SpiNNaker system to be able to correctly perform the required simulations.

- Sergio Davies, Alexander Rast, Francesco Galluppi and Steve Furber
  "*A forecast-based biologically-plausible STDP learning rule*"
  published in proceedings of the 2011 International Joint Conference on Neural Networks (IJCNN), July 2011, pages 1810–1817 (Davies et al., 2011a).
  This paper forms part of chapter 6.

- Sergio Davies, Javier Navaridas, Francesco Galluppi and Steve Furber
  "*Population-Based Routing in the SpiNNaker Neuromorphic Architecture*"
  published in proceedings of the 2012 International Joint Conference on Neural Networks (IJCNN), July 2012, pages 1932–1939 (Davies et al., 2012b). This paper is included as chapter 5 of this thesis.

- Xin Jin, Alexander Rast, Francesco Galluppi, Sergio Davies and Steve Furber
  "*Implementing Spike-Timing-Dependent Plasticity on SpiNNaker Neuromorphic Hardware*"
  published in proceedings of the 2010 International Joint Conference on Neural Networks (IJCNN), July 2010, pages 2302–2309 (Jin et al., 2010c). This paper is the background for the learning algorithm presented in chapter 6.

- Francesco Galluppi, Sergio Davies, Alexander D. Rast, Thomas Sharp, Luis A. Plana, Steve B. Furber
  "*Hierarchical Configuration System for a Massively Parallel Neural Hardware Platform*"
  published in proceedings of the ACM International Conference on Computing Frontiers 2012 (CF'12), June 2012, pages 183–192 (Galluppi et al., 2012a). This paper contains the description of the current system used to map neural networks on the SpiNNaker system. The topic of this paper is discussed in paragraph 3.5.

The author of this thesis has also contributed to the development and writing of the papers listed below. However, these papers will not be included directly in this work.

- Xin Jin, Francesco Galluppi, Cameron Patterson, Alexander Rast, Sergio Davies, Steve Temple and Steve Furber
  "*Algorithm and Software for Simulation of Spiking Neural Networks on the Multi-Chip SpiNNaker System*"
  published in proceedings of the 2010 International Joint Conference on Neural Networks (IJCNN), Jul 2010, pages 649–656 (Jin et al., 2010a);

- Francesco Galluppi, Alexander Rast, Sergio Davies, and Steve Furber
  "*A general-purpose model translation system for a universal neural chip*"
  published in proceedings of the 17th international conference on Neural information processing: theory and algorithms - Volume Part I, ICONIP'10, pages 58–65 (Galluppi et al., 2010);

- Alexander Rast, Francesco Galluppi, Sergio Davies, Luis A. Plana, Thomas Sharp and Steve Furber
  "*An Event-Driven Model for the SpiNNaker Virtual Synaptic Channel*"
  published in proceedings of the 2011 International Joint Conference on Neural Networks (IJCNN), Jul 2011, pages 1967–1974 (Rast et al., 2011b);

- Andrew Webb, Sergio Davies and David Lester
  "*Spiking Neural PID Controllers*"
  published in Lecture Notes in Computer Science, 2011, Volume 7064/2011, 18th International Conference on Neural information processing, ICONIP 2011, Nov 2011, pages 259–267 (Webb et al., 2011);

- Francesco Galluppi, Sergio Davies, Terry Stewart, Chris Eliasmith and Steve Furber "*Real Time On-Chip Implementation of Dynamical Systems with Spiking Neurons*"
  published in proceedings of the 2012 International Joint Conference on Neural Networks (IJCNN), July 2012, pages 2455–2462 (Galluppi et al., 2012b).

1.7 Workshops, conferences and invited talks

During the course of his Ph.D. the author has attended a number of events.

Workshops:

- 25th April 2010 - 8th May 2010 — CapoCaccia cognitive neuromorphic engineering workshop — Capocaccia, Sardinia, Italy;

- 27th June 2010 - 14th July 2010 — Telluride neuromorphic cognition engineering workshop — Telluride, Colorado, U.S.A. The outcome of this workshop has been described in the paper (Davies et al., 2010);

- 6th May 2011 - 14th May 2011 — CapoCaccia cognitive neuromorphic engineering workshop — Capocaccia, Sardinia, Italy;

- 26th June 2011 - 16th July 2011 — Telluride neuromorphic cognition engineering workshop — Telluride, Colorado, U.S.A. The outcome of this workshop has been described in the paper (Galluppi et al., 2012b);

- 12th September 2011 - 13th September 2011 — FACETS-ITN neuromorphic systems workshop — Heidelberg, Germany;

- 16th April 2012 - 17th April 2012 — SpiNNaker hands-on meeting — Southampton, U.K.;

- 29th April 2012 - 13th May 2012 — CapoCaccia cognitive neuromorphic engineering workshop — Capocaccia, Sardinia, Italy;

- 1st July 2012 - 21st July 2012 — Telluride neuromorphic cognition engineering workshop;

Conferences:

- 18th July 2010 - 23rd July 2010 — International Joint Conference on Neural Networks (IJCNN) 2010 held within the World Congress on Computational Intelligence (WCCI) 2010 — Barcelona, Spain; Presentation of the work described in the paper (Jin et al., 2010c);
- 22nd November 2010 - 25 November 2010 — International Conference On Neural Information Processing (ICONIP) 2010 — Sydney, Australia. Presentation of the work carried out during the Telluride 2010 workshop and described in the paper (Davies et al., 2010).
- 3rd May 2011 - 5th May 2011 — ACM Computing Frontiers conference — Ischia, Campania, Italy. Presentation of the work described in the paper (Davies et al., 2011b)
- 31st July 2011 - 5th August 2011 — International Joint Conference on Neural Networks (IJCNN) 2011 — San Jose, California, U.S.A.. Presentation of the paper on the STDP-TTS learning algorithm described in (Davies et al., 2011a);
- 10th June 2012 - 15th June 2012 — International Joint Conference on Neural Networks (IJCNN) 2012 held within the World Congress on Computational Intelligence (WCCI) 2012 — Brisbane, Australia; Presentation of the work described in the paper (Davies et al., 2012b);

Invited talks:

- 18th July 2011 — Invited talk at University of California S. Diego — California, U.S.A.;
- 19th July 2011 — Invited talk at Qualcomm and Brain Corporation — S.Diego, California, U.S.A.;
- 20th July 2011 — Invited talk at Salk institute — S.Diego, California, U.S.A.;
- 28th July 2011 — Invited talk at University of Stanford — California, U.S.A.;

1.8 Structure of the thesis

This thesis comprises 7 chapters, including this one.

Chapter 2

The second chapter presents the division into categories of neural network simulators primarily from a theoretical point of view, and then considers some of the simulators which have been developed around the world.

Chapter 3

The third chapter explains thoroughly the whole SpiNNaker project, the environment in which the work described in this thesis has been developed.

Chapter 4

The fourth chapter presents the first contribution of this research: the SpikeServer. This is the real-time input/output spike interface for the SpiNNaker board. This branch of research started during the Telluride Neuromorphic Cognition Engineering Workshop 2010 when a real-time silicon retina was interfaced with the SpiNNaker board (Davies et al., 2010). The architecture and the results collected from the SpikeServer follow the author's research. Cameron Patterson contributed through discussions on the tests to perform and in helping to write the paper.

Chapter 5

The fifth chapter presents the second contribution of this research: the population-based routing technique. Neurons are no longer connected one by one, but populations of neurons are connected using the same "channel" (which represents the nerve bundle in biological neural networks). Javier Navaridas contributed to this work describing the routing algorithms for multicast packets which are known in the literature; on the basis of this knowledge the author designed, implemented and tested the algorithm, Francesco Galluppi contributed to this work in the software testing phase. The paper (Davies et al., 2012b) on this topic was mainly written by the author with a contribution on the literature review on known algorithms by Javier Navaridas and proof-reading by Francesco Galluppi.

Chapter 6

The sixth chapter presents the main contribution of this thesis: the STDP-TTS learning rule. Theoretical contributions to this rule were given both by Alexander Rast and Francesco Galluppi. Alexander Rast, in particular, contributed to the paper writing phase while Francesco Galluppi helped in running network simulations on SpiNNaker and on Matlab.

Chapter 7

The final chapter discusses and summarizes the research undertaken so far and its possible evolution in the future.

Chapter 2

Neural network simulators

2.1 Introduction

This chapter presents an introduction to the tools and strategies used in the simulation of Spiking Neural Networks (SNN). The description gives, first, a theoretical overview on a possible way of categorizing neural network simulators. The second part of the chapter then focuses on the main simulators developed in the field and describes them relative to the categorization scheme proposed. This chapter gives the reader the grounds for a critical comparison between the various simulators developed in the field and the one in which this thesis work has been carried out: the SpiNNaker system. The SpiNNaker project is described thoroughly in the next chapter.

2.2 Categorization of SNN simulators

In this section a categorization of the SNN simulators is proposed. The first characteristic splits the whole class in two halves: hardware and software simulators.

Hardware simulators are those which include the development of dedicated hardware (e.g. a Printed Circuit Board, a special-purpose chip, etc.).

Software simulators are those developed to run on a standard computational unit (e.g. a Personal Computer, a computer cluster, etc.).

In addition there are hybrid simulators which require both the development of custom hardware and software to complete the simulator (e.g. FPGA device, custom built chip including a computational unit, etc.). Since, for this class of simulators, the development of new hardware is required to provide a computational substrate for the software simulator, they can be classified as hardware neural network simulators.

2.2.1 Hardware simulators

Hardware simulators are those which include the development of dedicated hardware to simulate SNN. This class includes simulators with specific software which runs on dedicated hardware.

This class comprises analogue and digital hardware simulators, differentiated by how the neurons are implemented: if neurons are implemented directly with analogue components (transistors, capacitors, resistors, etc.) then it is an analogue hardware simulator.

Alternatively, if the simulator is based on digital circuits which run dedicated software of a specific neuron model, and the values of the physical quantities can assume only discrete values, then this is a digital hardware simulator.

The literature also describes hardware projects which involve a combination of the two techniques. Often the distribution of the spikes across a system may use digital circuits, while the neural model is simulated using analogue components. In this case the technique used to build the circuit which emulates the neural model is used as reference to identify the type of simulator.

Both these classes can be subdivided on the basis of the number of neuron models which the simulator is able to run: single and multiple neuron models. Single neuron model simulators are those which, besides the reconfigurability of the synapses and the variability of neuron parameters, permit only one specific neuron model to be run (e.g. leaky integrate-and-fire, adaptive exponential leaky integrate-and-fire, etc.). Multiple neuron model hardware simulators are those which permit different neuron models to run on the same hardware in a reconfigurable network. This means that the same chip permits different neuron models to run in different runs of a simulation.

While both analogue and digital simulators are able to run a single neuron model simulation, only digital hardware simulators are likely to be able to simulate multiple neuron models; in digital hardware it is easy to modify the part connected to the computation of the state variable update of the model developed (e.g. modifying the connection between multipliers, adders and other basic components), while in analogue hardware it is very hard to reconfigure the circuitry (e.g. transistors, capacitors and resistances) to a second neuron model.

Over all these taxonomies of simulators there is a relation with time: the simulation can be in continuous time (analogue hardware), in discrete time (analogue and digital hardware) or in abstract time (digital hardware) (Rast, 2010).

Continuous and discrete time simulations are generally well-known paradigms.

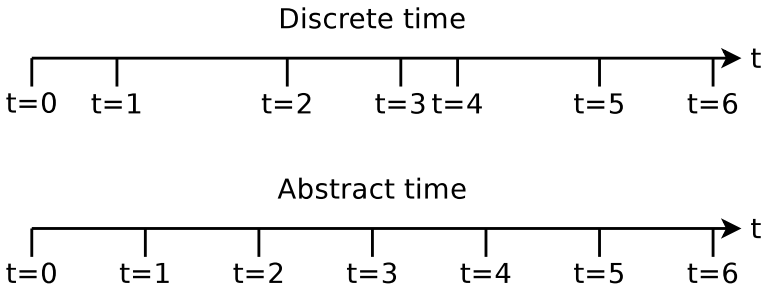


Figure 2.1: Difference between abstract-time simulator and discrete-time simulator.

Simulators running in continuous time have the physical values of the simulation always meaningful. Discrete time simulators have the time divided into "events" in which the values of the simulation are meaningful. An abstract time simulator has temporal slots in which the boundaries have a definition in the real life time. Between these boundaries the computation takes place to move the simulation one step forward integrating the differential equation(s) across one time step. The difference between the abstract time simulator and the discrete time simulator can be described through Fig.2.1: a discrete time simulator presents computation intervals which may not be constant in time. An abstract time simulator has, instead, time-constant computation intervals. Therefore, a discrete time simulator may become an abstract time simulator if the computation interval is kept constant in all the slots.

Time relation is a characteristic that transcends the categorization of the simulators. These, in fact, can be classified with respect to biological time (Fig.2.2):

- Real-time simulators: the time of the simulation corresponds to biological time. Therefore one millisecond of simulation corresponds to one millisecond in biological time. This is a strict constraint.
- Accelerated time: the time of the simulation runs faster than biological time. Therefore in one millisecond the simulation will model more than one millisecond of activity of the biological model.
- Non real-time simulators: the time of the simulation runs slower than biological time. Therefore in one millisecond the simulation will model less than one millisecond of activity of the biological model.

A hierarchical definition of the classes of hardware simulator is graphically described in Fig.2.3.

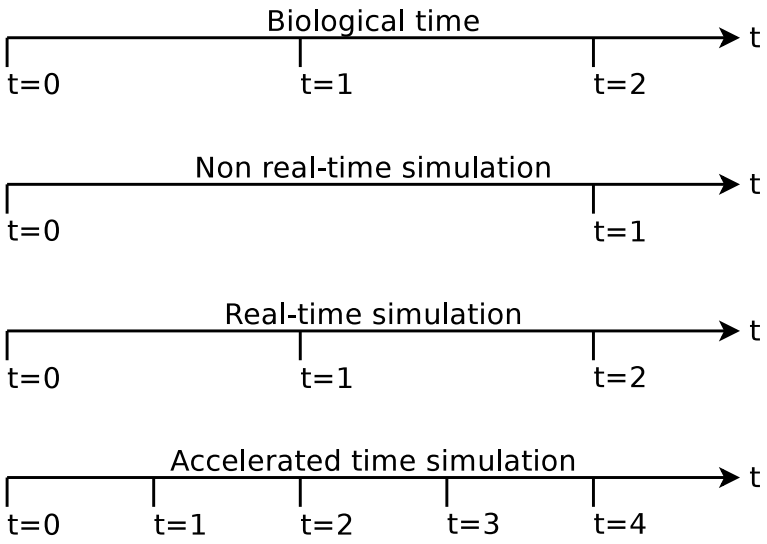


Figure 2.2: Relation between simulation time and biological time.

2.2.2 Software simulators

Software simulators are those developed to run on a standard computational unit (e.g. a Personal Computer, a computer cluster, etc.).

The neuron model implemented in these type of simulators is represented by some lines of code which implement the mathematical model (in terms of Ordinary Differential Equations - ODE) of a biological neuron. Since the neuron model here is implemented in software, it is very likely that those simulators are able to simulate multiple neuron models (perhaps even within the same run of a simulation). However it is possible that simulators implement only single neuron model, in which case, the simulator has usually been strongly optimized for a specific neuron model.

Software simulators can be clock driven (synchronous simulators) or event driven (asynchronous simulators) and always run in discrete time (Brette et al., 2007) or in abstract time (Rast, 2010).

Also, in this case, there is a time relationship which transcends the categories of simulator presented until now. Usually the more neurons which are simulated, the slower is the simulation, because the simulator substrate has a finite computational power shared between all the neurons: the greater the number of neurons simulated, the greater the computation demands of the simulator and hence the greater time taken to perform a time step in the simulation. Eventually, for medium scale neural networks, the time relation goes below the real-time boundary (in other words, it becomes slower

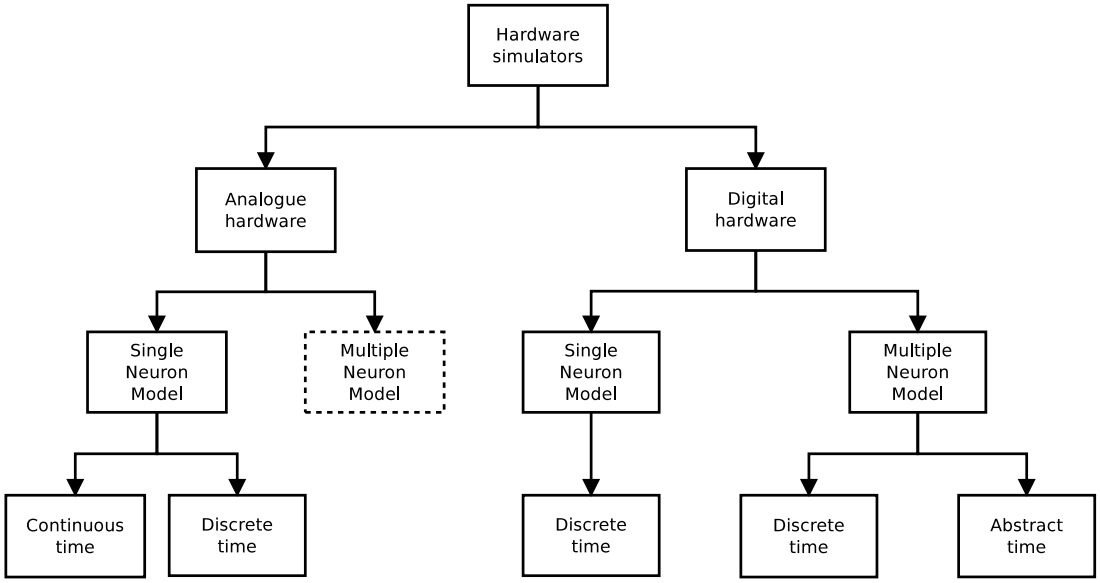


Figure 2.3: Hierarchical representation of hardware neural network simulators

than real-time).

A hierarchical definition of the classes of software simulator is graphically described in Fig.2.4.

2.2.3 Learning in modern simulators

All classes of simulator (both hardware and software) described can incorporate learning functions which is a characteristic which defines the architecture of neural network simulators.

A simulator may be characterized by one or by multiple learning rules, and some of them may even run during the same simulation (e.g. Short-Term Plasticity and Spike Timing Dependent Plasticity).

However, for the purposes of this dissertation, simulators are classified on the basis of the presence or absence of learning capabilities, without detailing the learning rules implemented.

2.3 Review of main simulators developed

In this section a review of some existing simulators is presented; these have usually been developed as part of neuroscientific projects. To reflect the differentiation made in the previous section, here, hardware and software simulators are presented separately.

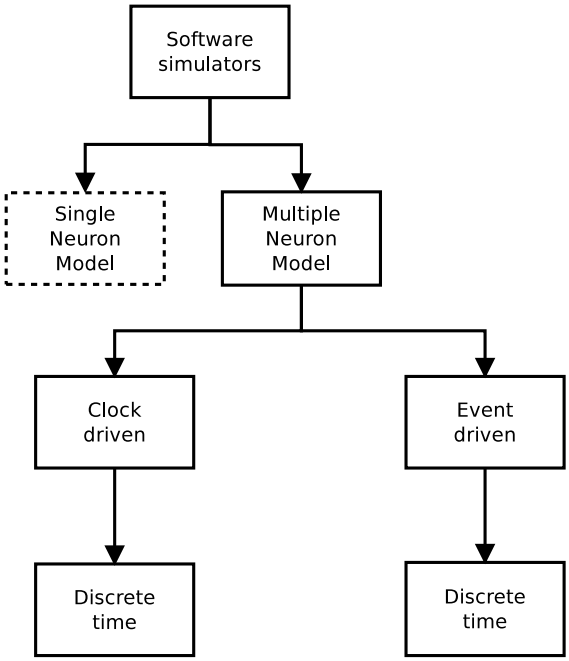


Figure 2.4: Hierarchical representation of software neural network simulators

However, this section presents only a summary of the reviewed simulators. Complete descriptions of each simulator are available in the cited references. Moreover, more complete reviews have been published in numerous articles (e.g. Brette et al. (2007), Misra and Saha (2010), Draghici (2000) and Zhu and Sutton (2003)).

2.3.1 Hardware

SyNAPSE

"SyNAPSE" is the acronym of the System of Neuromorphic Adaptive Plastic Scalable Electronics project. The goal of this project is to design a neuromorphic chip which is able to replicate a mammalian brain in size, functionality and power consumption: it should be able to recreate 10^{10} neurons with 10^{14} synapses consuming $1KW$ of electrical power and occupying $2dm^3$ (liters) of space (Seo et al., 2011; Ananthanarayanan et al., 2009).

The simulator uses digital components to simulate a leaky integrate-and-fire neuron model. Spikes are transmitted from one neuron to subsequent ones using a communication crossbar (Merolla et al., 2011).

Four different chips have been produced (Seo et al., 2011):

1. A base design chip with binary synapses and "standard" leaky integrate-and fire

neurons;

2. A slim neuron variant, which fixes spiking, learning parameters and the structure of the network in a two-layer learning network;

3. A 4-bit synapse variant which allows modification of synaptic weights in a way closer to biology;

4. A low leakage variant to reduce power dissipated by neurons. This has some cost in terms of the minimum operating voltage of the memory array.

In summary, this project can be classified as a hardware, digital, real-time simulator with learning capabilities. The neural model is integrated over steps of $0.1msec$ with a discrete time paradigm.

Blue Brain Project

The Blue Brain Project aims to provide a computational substrate for molecular-level simulations that present biological realism. The goal of this platform is to "*simulate the brains of mammalians with a high level of biological accuracy and, ultimately, to study the emergence of biological intelligence*" (Markram, 2006).

The platform "Blue Gene/L" is provided by IBM and comprises $8,192$ CPU nodes each being a PowerPC 440 running at 700 MHz, with a peak performance of 22.4 TFLOPS and 2 TB of memory (Markram, 2006).

As described before, the software running the simulation involves biological details, therefore this simulator can be classified as: hardware neural network simulator, running slower than real-time with a discrete time paradigm. The capabilities incorporated in this simulator involve learning, as well as other biological details.

FACETS - BrainScalesS

The FACETS (Fast Analog Computing with Emergent Transient States) project delivered wafer-scale integration of neuromorphic chips which simulate adaptive exponential leaky integrate-and-fire neurons (Schemmel et al., 2010). In addition it is possible to reconfigure the circuit such that the adaptation part can be deactivated and hence the neuron can behave as a standard leaky integrate-and-fire neuron. Short-term plasticity and long-term plasticity mechanisms are implemented on the synapses. The distribution of spikes uses digital components and digital interfaces may be used to route

spikes between two wafers or to a computer. The system runs 10^4 times faster than real-time, consequently, this simulator can be classified as multiple-model analogue hardware simulator with learning capabilities running faster than real-time.

The FACETS project is completed and the outcome forms the basis of the current BrainScaleS project.

Neurogrid

The core of this simulator is a neuromorphic analogue chip simulating 256×256 leaky integrate-and-fire neurons in real-time (Silver et al., 2007). The distribution of spikes across the system uses digital interconnections which propagate spikes from one layer to the subsequent using an Address Event Representation (AER) protocol (Boahen, 2000). This protocol defines the transmission of neural events (action potentials) in simulators by sending the address of the element which had emitted it. In the Neurogrid project, an external FPGA is required to program arbitrarily neuron interconnectivity.

In summary, this is an single-model hardware simulator running in real-time without learning capabilities.

VLSI chips from Zurich

The chips built by the group at the Institute for NeuroInformatics (INI) in Zurich implement analogue leaky integrate-and-fire neurons with 28 plastic and 4 non-plastic synapses per neuron, running in real-time (Indiveri et al., 2009). The learning rule implemented in hardware is related to the standard STDP learning rule, but uses bi-stable state synapses: one state provides a high synaptic weight, while the other state provides a low weight. The distribution of the spikes in the architecture takes advantage of the AER protocol.

This simulator can be classified as a single-model, analogue, hardware simulator with learning capabilities running in real-time.

2.3.2 Software

Brian

Brian is a software neural simulator written in Python (Goodman and Brette, 2008). It is able to simulate multiple neuron models, but much slower than real-time, especially when simulating complex neural networks. It is a clock-driven simulator, where all

events take place on a fixed time grid ($t = 0, dt, 2dt, 3dt, \ldots$). Learning features are available for the simulation.

In summary, this is a software simulator which allows multiple neuron models in the same simulation and implements learning features. It uses a discrete, clock-driven, time paradigm, running slower than real-time.

Neuron

Neuron is a software simulator for creating and using models of biological neurons and neural circuits (Brette et al., 2007). It is supported by a complete development environment to describe characteristics of neurons and neural circuits. To advance simulations in time, users have a choice between built-in clock-driven methods (a backward Euler and a Crank-Nicholson variant both using fixed time step) and event-driven methods (fixed or variable time step which may be system-wide or local to each neuron, with second order threshold detection).

In summary, this is a software simulator which allows multiple neuron models in the same simulation and implements learning features. It uses a discrete clock-driven and event-driven time paradigm, running slower than real time.

Nest

The purpose of the simulator Nest was to be the reference implementation to support the development of neural network simulators (Brette et al., 2007). The networks this simulator is able to run can easily grow up to 10^5 neurons and beyond, with realistic connectivity.

This simulator supports heterogeneity of neurons and synapses in a single simulation. It implements both a global time-driven simulation mechanism and an event-driven algorithm, so that spikes are not fixed on the discrete grid.

In summary, Nest is a software simulator which allows multiple neuron models in the same simulation and implements learning features. It uses a discrete clock-driven and event-driven time paradigm, running slower than real time.

Genesis

Genesis (Wilson et al., 1989) (GEneral NEural SImulation System) is a simulator that aims to reproduce the biological behaviour of neural systems with details ranging from biochemical reactions to large scale neural networks.

Genesis was the first simulator able to cope with large scale neural networks and the main application is connected with the simulation of biological neural systems.

In summary Genesis is a software simulator which allows multiple neuron models in the same simulation and implements learning features.

SpikeFun - DigiCortex

This project aims to build a large-scale biologically-realistic neural network simulator for a standard PC (Dimkovic, 2011). The simulator engine is called DigiCortex, while the graphical user interface is called SpikeFun. Currently it implements only the Izhikevich neuron model with 30 compartments per neuron and AMPA, GABA and NMDA synapses; learning capabilities are based on the standard STDP rule. The simulator uses a discrete time paradigm and its speed depends on the number of neurons simulated. However, even for small scale simulations, the speed of the simulation appears to be slower than real-time.

In summary this project can be classified as a software simulator, running a single neuron model in discrete time with learning capabilities.

NEF - Nengo

The Neural Engineering Framework (NEF) is a mathematical background which allows the use of neural networks in the field of control theory (Eliasmith and Anderson, 2004). Using values encoded as neuron spiking rates it is possible to evaluate functions (even non-linear) with the purposes of computing more complex algorithms. Nengo is the software simulator which implements the principles of the NEF. The simulator uses one main neural model, the leaky integrate-and-fire neuron model, and sets of encoders and decoders to represent numeric values. Numeric values are encoded as collection of spike rates for each population of neurons. The number are converted into spike rates by an encoder population, and backward by a decoder population. The principles of the Neural Engineering Framework allow the possibility of synaptic plasticity with the purpose of training the network to perform a particular function. The learning paradigm featured in this case is, probably, supervised training with a teaching signal which performs error correction.

In summary, this simulator allows multiple neuron models, runs slower than real-time, has a discrete clock-based time paradigm, and incorporates learning features.

2.3.3 SpiNNaker

The SpiNNaker system is presented thoroughly in the next chapter, however here we give a summary of its capabilities to allow the comparison with the other spiking neural network simulators. This architecture is a real-time simulator running on an event-driven abstract-time paradigm using multiple neural models during the same simulation, and incorporating learning capabilities. As shown in the comparison table 2.1, the SpiNNaker system provides some features that have not been proposed by other simulators. For each simulator, the features developed have been marked with a tick. Only simulators including software development have been categorised using the "Event-driven" and "Clock-driven" classes. For Hardware simulators the two categories are marked as "Non Addressable".

2.4 Summary

In this section a possible categorization of modern spiking neural network simulators has been presented; these are classified on the construction methodology and the features implemented, as specified in the summary table 2.1.

The next chapter introduces the SpiNNaker simulator, its design and features, categorizing it using the methodology presented in the current chapter.

Simulator name	Hardware	Software	Analogue	Digital	Single neural model	Multiple neural model	Learning capabilities	Continuous time	Discrete time	Abstract time	Clock-driven	Event-driven	Accelerated time	Real time	Non real time
Synapse	✓			✓	✓		✓		✓		N.A.	N.A.		✓	
Facets - BrainscaleS	✓		✓			✓	✓	✓			N.A.	N.A.	✓		
Neurogrid	✓		✓		✓			✓			N.A.	N.A.		✓	
VLSI from Zurich	✓		✓		✓		✓	✓			N.A.	N.A.		✓	
Brian		✓		✓		✓	✓		✓		✓				✓
Neuron		✓		✓		✓	✓		✓		✓	✓			✓
Nest		✓		✓		✓	✓		✓		✓	✓			✓
Genesis		✓		✓		✓	✓		✓		✓	✓			✓
SpikeFun		✓		✓	✓		✓		✓		?	?			✓
NEF - Nengo		✓		✓		✓	✓		✓		✓				✓
Blue Brain Project	✓	✓		✓		✓	✓		✓		?	?			✓
SpiNNaker	✓	✓		✓		✓	✓			✓		✓		✓	

Table 2.1: Comparison between features of spiking neural network simulators. Question marks identify features that were not easily possible to associate with the particular simulators.

Chapter 3

SpiNNaker project overview

3.1 Introduction

In this chapter the SpiNNaker project is introduced. The topics presented are an overview of various aspects of the whole project and form the basis on which the contribution presented in the following sections has been carried out. More detailed description of the technical challenges and contributions made during the development can be found in the articles cited throughout the description.

3.2 Description of the SpiNNaker project

The acronym SpiNNaker stands for Spiking Neural Network Architecture (Furber et al., 2006) which is a hardware-based, real-time, universal, neural network simulator following an event-driven computational approach (Furber et al., 2012; Rast et al., 2010c). This project involves the design of a chip and the development of dedicated software to simulate neural networks (Jin et al., 2008). This system tries to mimic the features of biological neural networks in various ways:

- **Native parallelism**: Each biological neuron is a primitive computational element within a massively parallel system. Likewise, SpiNNaker uses parallel computation;
- **Spiking communications**: In biology, neurons communicate through spikes. The SpiNNaker architecture uses source-based AER packets to transmit the equivalent of neural signals (i.e. action potentials). Each Address Event Representation (AER) packet identifies the event source through an addressing scheme.

The time of the event is intrinsically identified by the time of the packet itself.

- **Event-driven behaviour**: Neurons are very power efficient, and consume much less power than modern hardware. To reduce power consumption, the hardware is put into "sleep" state when idle, awaiting an interrupt (Jin et al., 2008);
- **Distributed memory**: In biology, neurons use only local information to process incoming stimuli. The SpiNNaker architecture features a hierarchy of memories: memory local to each of the cores and an SDRAM local to each chip;
- **Reconfigurability**: In biology, synapses are plastic. This means that neural connectivities change both in shape and strength. The SpiNNaker architecture allows on-the-fly reconfiguration.

3.3 Hardware

3.3.1 Chip architecture

The core of this simulator is the SpiNNaker chip (Furber, 2011): a full-custom ASIC chip with 18 ARM 968 cores, running at $\approx 200MHz$ with low power consumption specifications and extended instruction set for digital signal processing. No floating-point unit has been embedded in the architecture of this chip to comply with the low power specifications and the space available on the die. Consequently, all computation implemented in the software must be based only on fixed-point operations. Additionally, the division operation is not part of the instruction set of the ARM architecture, and therefore must be implemented in software, or avoided if possible. Figure 3.1 describes the chip by functional blocks, and Figure 3.2 labels each major component on the die.

A full-custom router (Plana et al., 2007) stands between the cores and the input/output links where it can receive network packets from any source and route network packets to their correct destination(s). A Network on Chip (system NoC) interfaces the cores with the peripherals: System RAM, System ROM, MII (Ethernet) interface, Watchdog, System controller, PLLs and PL340. There are three types of memory available to each core:

- **Tightly Coupled Memory (TCM)** which is part of each ARM core and is divided in two parts: instruction TCM (ITCM) which is 32 KB and data TCM

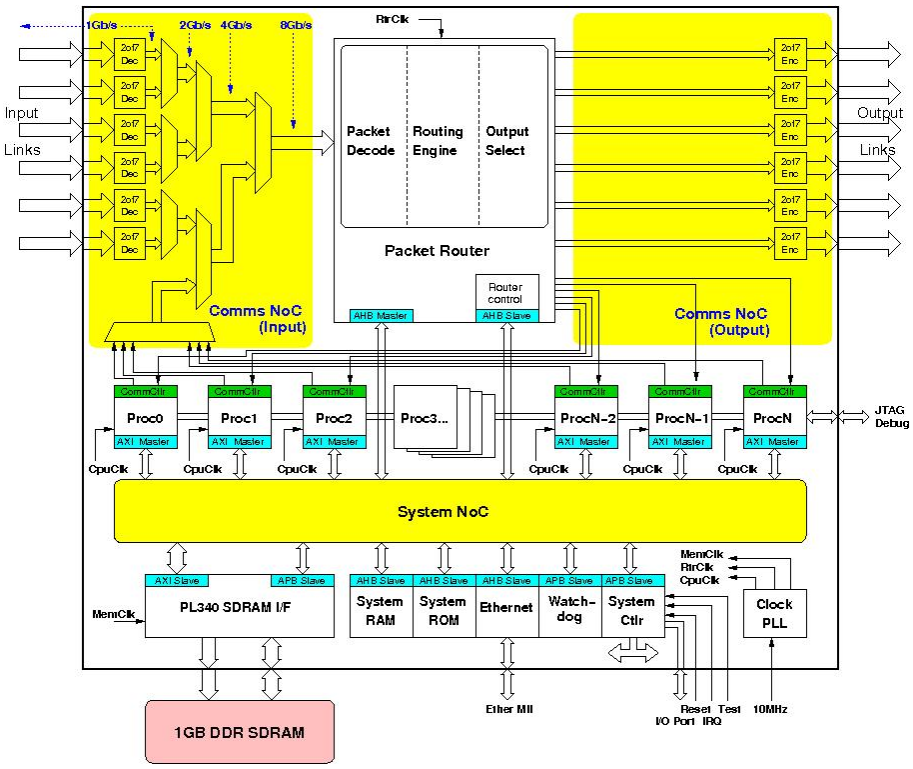


Figure 3.1: Block diagram of the full SpiNNaker chip

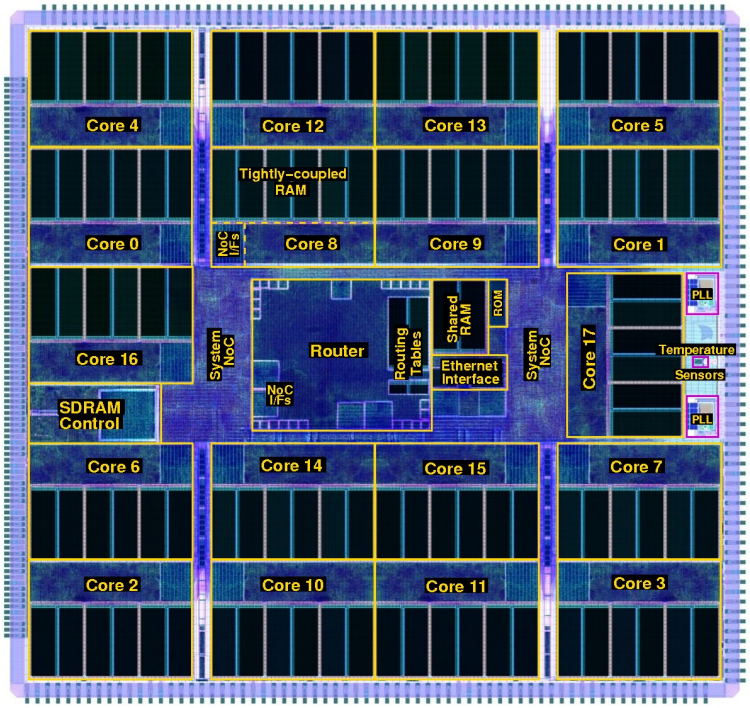


Figure 3.2: Layout of the SpiNNaker chip with labels identifying each functional block.

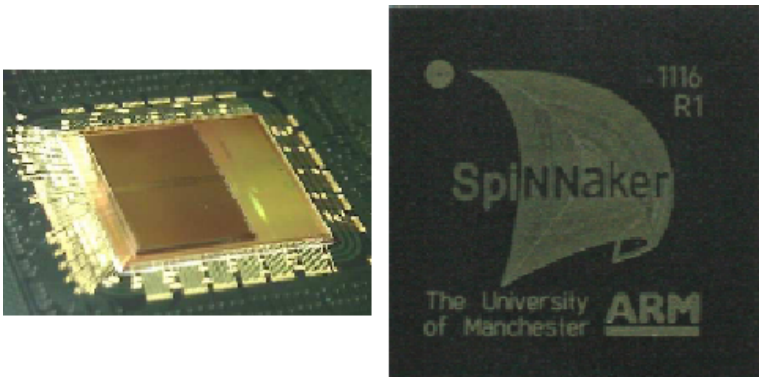


Figure 3.3: SpiNNaker chip package: the die of the SDRAM memory chip is mounted on the top of the SpiNNaker chip.

(DTCM) which is 64 KB. This memory is integrated into each core, and therefore each core accesses its own TCM;

- **System RAM**, which is integrated into the SpiNNaker chip and shared between all the processors. Its size is 32KB;
- **SDRAM**, which is a mobile laptop memory chip external to the SpiNNaker chip and accessible through the PL340 interface, shared between all the cores. The size of this memory is 128 MByte (identified with 1Gbit in Fig.3.1). Physically the die of the memory chip has been mounted on top of the die of the SpiNNaker chip, with bonding wires which connects the two dies internally to the packaging, as shown in Fig.3.3.

In addition there is a ROM memory shared between all the processors which contains the bootstrap software.

3.3.2 Chip interconnections

The SpiNNaker chip has six external links, as described in Figure 3.1, to connect to six other SpiNNaker chips in a 2-dimensional network of up to 256×256 chips (see Fig.3.4). The extremities of such a grid can be wrapped (see green links in Fig.3.4) to form a toroidal network, as depicted in Fig.3.5. Alternatively, it is possible, also, to visualize the 2-dimensional array of chips as a hexagonal-shaped network (see Fig.3.6).

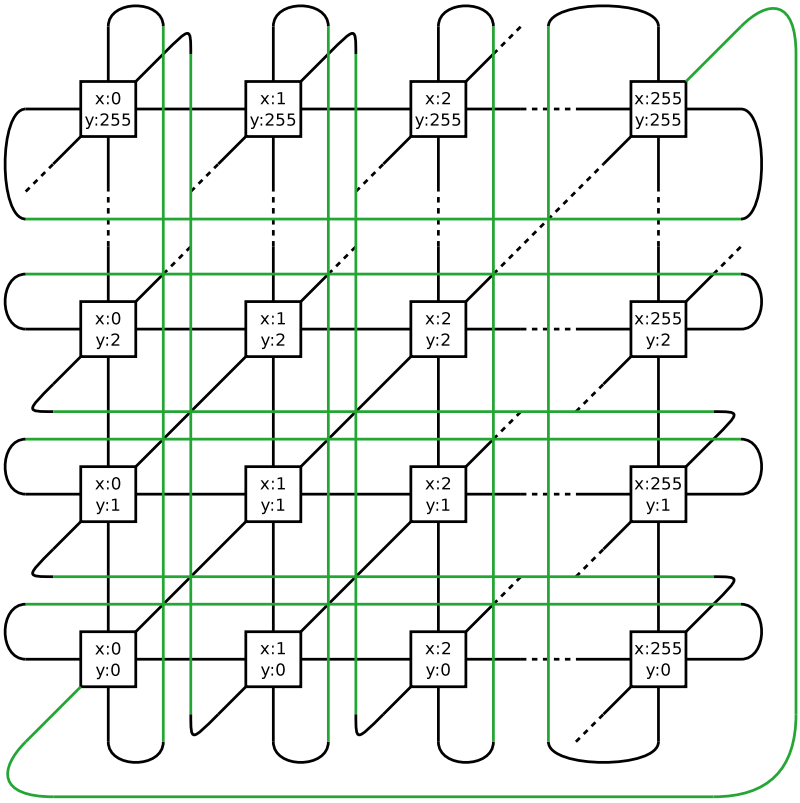


Figure 3.4: Two-dimensional grid of SpiNNaker chips with the needed connections (in green) to form the toroidal shape.

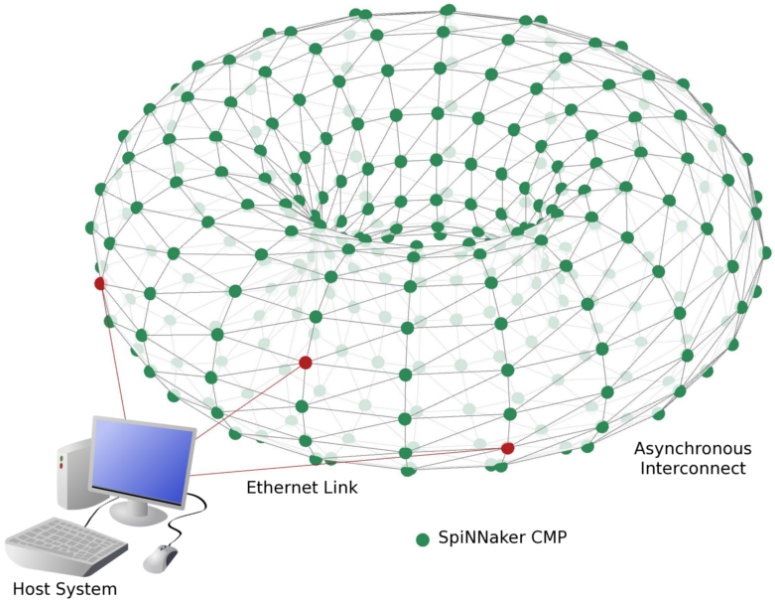


Figure 3.5: Appearance of the SpiNNaker chip network

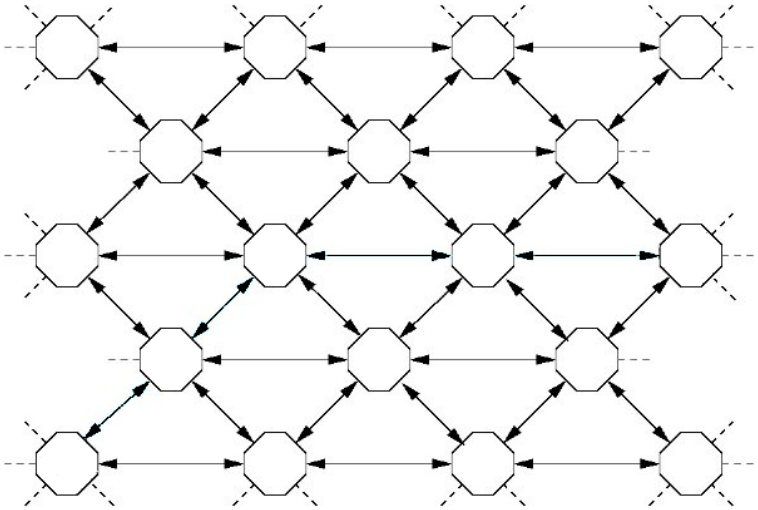


Figure 3.6: Hexagonal shaped SpiNNaker chip network

3.3.3 Interconnection router

The centre of the interconnection system, described earlier, is the router which is the main component of the SpiNNaker network (Wu et al., 2009; Wu and Furber, 2010). This part of the chip has been custom designed to deliver packets received to the appropriate destination(s) as fast as possible. The router is designed to handle four types of network packet which are routed between processors and/or chips:

- **Multicast (MC)** packets are designed to be sent towards multiple chips in the network. In neural application, this type of packet is used to send action potentials (i.e. spikes) to multiple destination chips and/or cores. The router stores a routing table that defines the way to handle this type of packet;
- **Point to point (P2P)** packets are routed towards a single destination chip in the network. Each chip, at the beginning of a simulation, is identified by a unique number (which is determined by its position in the network), and consequently populates the P2P routing table identifying the routes to all the possible destinations;
- **Nearest neighbour (NN)** packets are used to probe a neighbouring chip, to test if it is alive and, in case it does not respond, it is possible to access its memory directly through the router using “peek” and “poke” operations;
- **Fixed route (FR)** packets are directed towards the nearest Ethernet attached chip through a mechanism similar to the MC packets. For these packets the route is

specified in a router internal register.

Packets arriving at a router are presented one at a time and an internal arbiter is responsible for determining the order of packet processing. Each packet is processed independently of previously received packets (the router is stateless).

Multicast packets and routing table

Multicast packets are distributed by the router to one or multiple destinations at the same time. Each packet requires one clock cycle to be processed, so that the router, running at $\approx 200MHz$, is able to process $\approx 200,000,000$ packets per second. The Multicast packet router includes a routing table of 1024 entries which determines the route by which each network packet has to be distributed. The routing table comprises a ternary CAM (Content Addressable Memory) with three values per entry: a key value, a mask and a direction vector. The routing key of a received packet is compared with all the entries in the routing table simultaneously using a circuit such as that described in Fig.3.7. The field "Routing entry" is the value stored in the routing table, where the mask is also stored and is used to select which bits of the routing key should be used for the comparison. The field "Routing key" is the routing key of a packet received by the router which is being processed. The circuit which selects the routing entry, on the basis of the content of the fields "Routing entry" and "Mask" may have four different behaviours as illustrated in Tab.3.1.

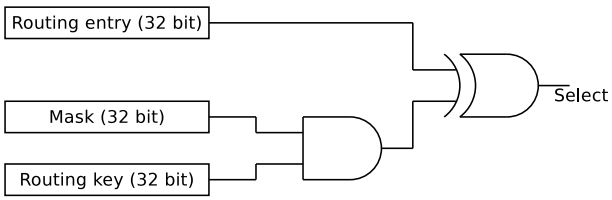


Figure 3.7: Circuit used to select a matching routing entry.

Routing entry	Mask	Matching condition
0	0	Match always
1	0	Match never
0	1	Match if the routing key value is 0
1	1	Match if the routing key value is 1

Table 3.1: Matching condition on the basis of the fields "Routing entry" and "Mask", as described in Fig.3.7

If one routing entry matches, the direction vector is used to determine where the packet is forwarded: it can be sent to any subset (including all) of the local processors and the output links (Furber, 2011). If multiple entries are matched, then the one that is stored in the lowest routing table location is used to determine the relay direction. The direction vector is a bit mask where each bit identifies one destination (see Tab.3.2). To relay the packet to multiple destinations, the direction vectors have to be OR-ed so that multiple "1"s identify all the desired destinations. Multicast packets have an additional (optional) field to carry 32 bits of information. So the total length (including 8 bit packet header) may be either 40 or 72 bits, depending on the presence or absence of the payload.

MultiCast direction vector (binary)	Output port	Direction
000000000000000000000001	Link 0	East
000000000000000000000010	Link 1	North-east
000000000000000000000100	Link 2	North
000000000000000000001000	Link 3	West
000000000000000000010000	Link 4	South-West
000000000000000000100000	Link 5	South
000000000000000001000000	Processor 0	Local
000000000000000010000000	Processor 1	Local
000000000000000100000000	Processor 2	Local
...	...	...
100000000000000000000000	Processor 17	Local

Table 3.2: Table of direction for multicast routing entries. The vectors are OR-ed to multicast the same packet to multiple directions

Point to point packets and routing table

Point to point (P2P) packets are sourced from one chip and directed towards another in the SpiNNaker network. When a chip receives a P2P packet, it checks in the routing table (specifically implemented for this type of packet - different from the multicast routing table) the direction to which to relay the packet. Since this type of packet is unicast, the Point to Point (P2P) router uses a direction field comprising three bits to determine to which output the packet should be routed (see Tab.3.3). If the destination is within the local chip, the packet is delivered to the monitor processor (as indicated by the Router Control Register).

The data that this type of packet is able to carry limits to 32 bits, therefore the total length of this packet (including the 8 bit packet header) is 72 bits.

P2P table entry	Output port	Direction
000	Link 0	East
001	Link 1	North-East
010	Link 2	North
011	Link 3	West
100	Link 4	South-West
101	Link 5	South
11x	Monitor processor	Local

Table 3.3: Table of directions for Point to Point routing entries

Nearest neighbour packets

The Nearest Neighbour (NN) router is used to send and receive information to/from neighbouring chips. This type of packet is used to initialize the system, to perform run-time flood-fill and for debug functions.

The packet sent from one chip on a specific link (or on all the links) is received by the chip on the other end of the link, which routes the packet to the monitor processor.

This type of packet can operate either with or without involving directly the cores in the chip: "Indirect" NN packets are directed to the monitor processor which will take action dependent on the content of the packet. "Direct" NN packets are processed by the router and the operations allowed are either "peek" or "poke" memory operations which can access directly resources on the system NoC.

This type of packet has an 8 bit header, 32 bits as memory address and 32 bits as data. If the required operation is a "peek", the packet length is 40 bits: 8-bit header and 32-bit memory address to read. The destination chip (or core) replies with a similar packet which contains 8-bit header and 32-bit data that is present at the required address. For a poke operation the packet is 72 bits long and includes all three fields.

Fixed route packets

This type of packet has a behaviour similar to that of the Multicast packet except that a single route is defined, independently of the content of the packet. The route is defined in a register so that each router directs this type of packet towards a single specific destination. The advantage of this type of packet is that it provides 64 bits of data to be transported to the destination (instead of the 32 bits provided by all the other packet types), which is helpful for software debugging purposes.

The length of this packet is always 72 bits.

3.3.4 External connections

The SpiNNaker chip has an MII interface to allow Ethernet connections to the outside world and is used to load data before simulation. During simulation it is used to retrieve the status of the system and of the neural network, to input stimuli to specific neurons and to get output spikes. At the end of the simulation it is possible to retrieve data on neurons and synapses, and to gather information stored about the simulation (e.g. spike times).

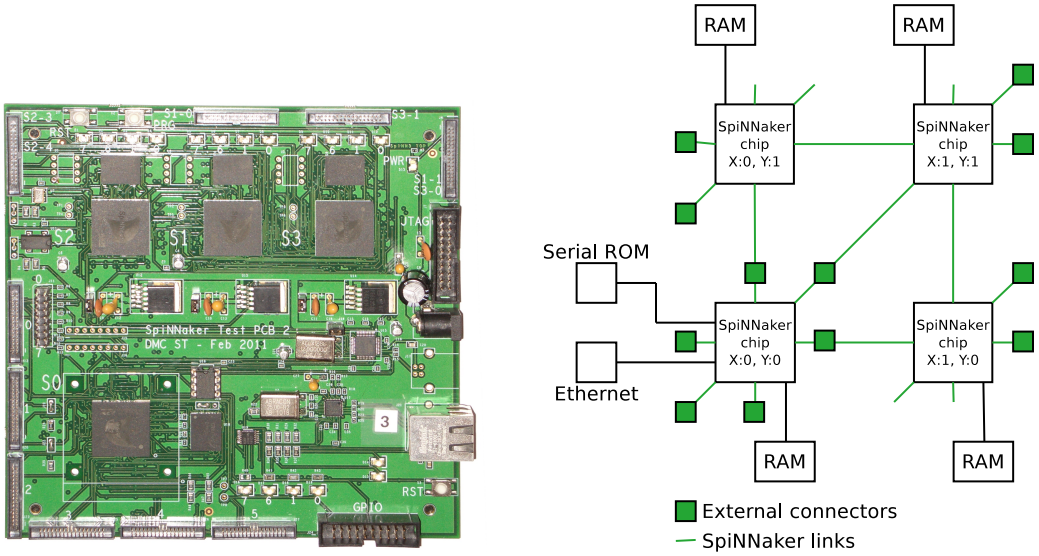
The external connection relies on UDP packet transfer over IP protocol. A SpiNNaker packet has been defined to send and receive information to/from all the chips and cores on the board. These packets are encapsulated into a UDP packet and include an instruction code, some parameters, if needed by the instruction, and a data field of 256 bytes.

3.3.5 Boards

Three types of board have been designed for the final version of the SpiNNaker chip. The Initial design included four chips on a board designed to provide external connections to other boards (see Fig.3.8(a) and block schematic Fig.3.8(b)).

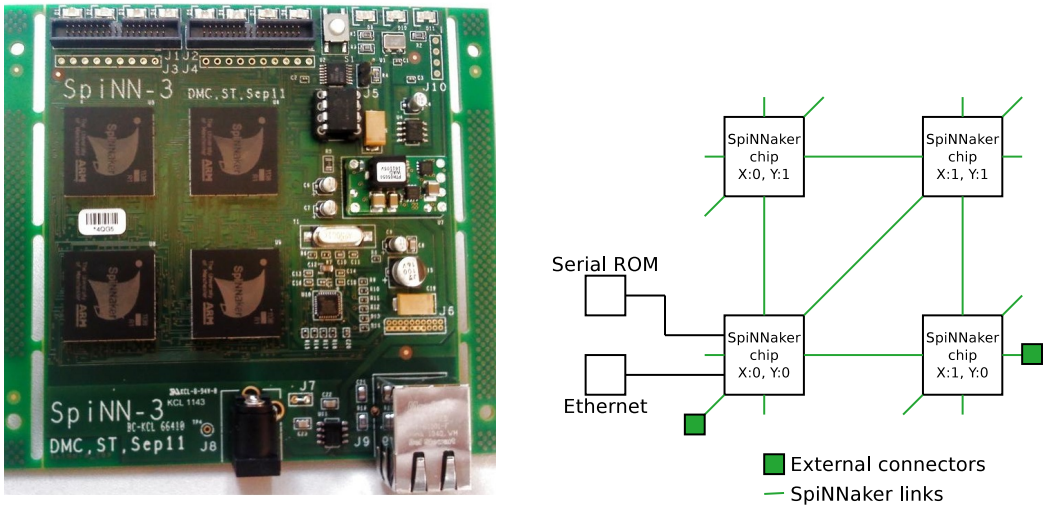
Subsequently, a smaller printed circuit board was designed with the purpose of including them in toy robots, and therefore this board has been nicknamed "Bunny-Board" (see Fig.3.9(a)): the difference from the previous version is in the number of external connections provided (see board block diagram Fig.3.9(b)), and the use of the in-package memory chip only.

Finally, a third board has been designed as the basic module of the final SpiNNaker system. This board mounts 48 SpiNNaker chips in a hexagonal-shaped network (see Fig.3.10(a) and block diagram Fig.3.10(b)). Multiple boards will be linked through SATA-like connectors (it is possible to see them in Fig.3.10(a) on the left and right edges of the board) to form the complete SpiNNaker system in the (virtual) shape of the toroid presented in Fig.3.5.



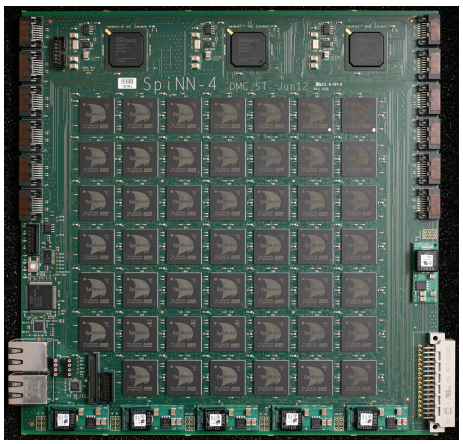
(a) Picture (b) Block Diagram

Figure 3.8: First release of the SpiNNaker board.

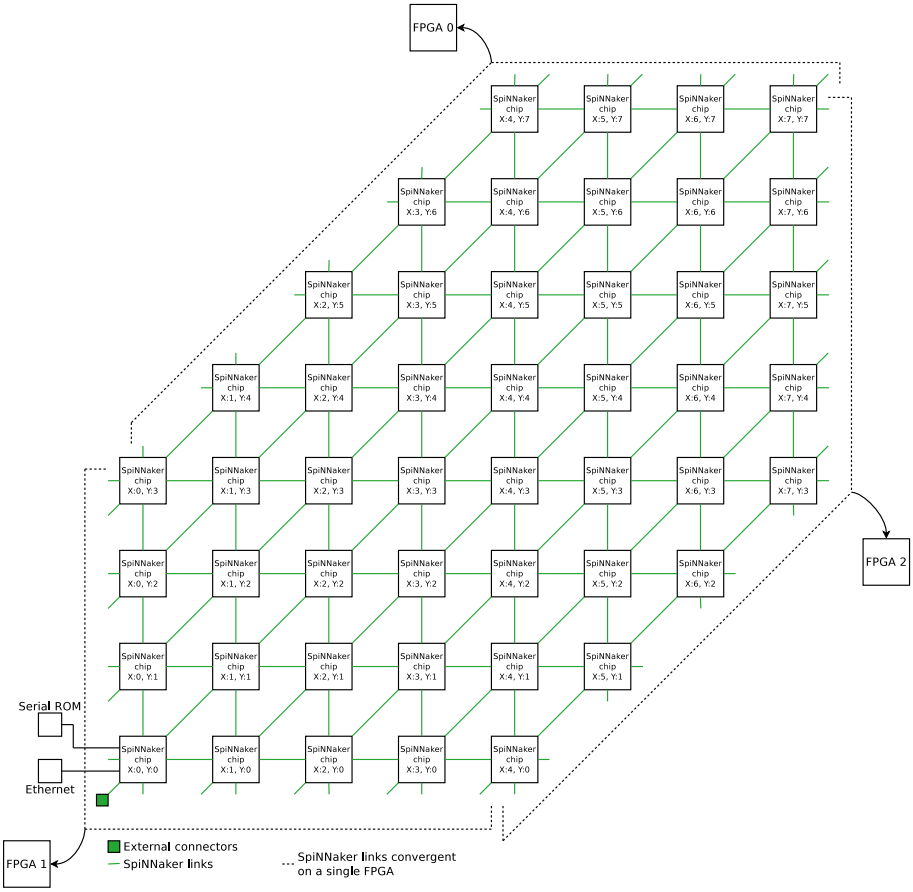


(a) Picture (b) Block Diagram

Figure 3.9: "BunnyBoard"



(a) Picture



(b) Block Diagram

Figure 3.10: 48-chip SpiNNaker board.

3.4 Software

The SpiNNaker simulator is a hybrid design which joins custom hardware with custom simulator software adapted to the underlying hardware. The software developed for the SpiNNaker system has been designed to maximise its power efficiency; whilst idle, all processors are kept in a low-power sleep mode with interrupts enabled. When an interrupt is received, the processor wakes up, performs the required actions to respond to the interrupt, and then returns to sleep. The three main interrupt sources are as follows:

1. **Timer interrupt**: at millisecond intervals (though this interval is programmable) the timer triggers an interrupt to update the membrane potential status of each neuron simulated by that core – integrating the Ordinary Differential Equation(s) of the neural model. Should a neuron reach its firing threshold, a network spike packet is emitted through the SpiNNaker network and routed to the core(s) on which the destination neuron(s) reside. This is described thoroughly in the next section;

2. **Packet received interrupt**: each time a packet is received from the SpiNNaker network a Direct Memory Access (DMA) operation is triggered to bring in the synaptic connection information for that neuron from the external SDRAM. Spikes received within a single millisecond time slot are considered as belonging to the same simulation millisecond;

3. **DMA complete interrupt**: this interrupt is triggered when the synaptic connection information has been transferred to the core's local memory for processing. Using a circular buffer to emulate the synaptic and axonal delays, at the appropriate time the synaptic weight is used to calculate the current injected into the post-synaptic neuron(s). The handler of the last two interrupts is described thoroughly in section 3.4.2.

The number of neurons which each core is able to simulate within the timing constraint varies with the computational complexity of the neuron model, and with the connectivity pattern required (Rast et al., 2010a). An upper bound on the maximum number of neurons that each core is able to simulate is imposed by the design specifications of the SpiNNaker system itself. All components, in particular those related to memory access and communication, have been designed to support the biologically

plausible traffic generated by up to 1,000 real-time spiking neurons per core. The software is mainly written in C and can be divided into two main parts detailed in the next two sections.

3.4.1 The neuron simulator

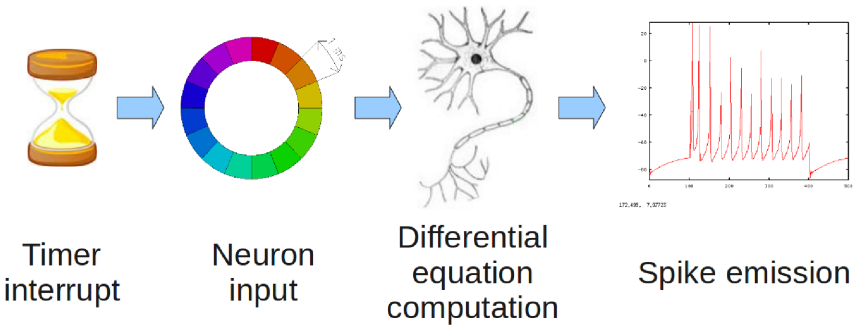


Figure 3.11: Block description of SpiNNaker neuron simulator software

The first part of the software is the neuron simulator described by the functional blocks in Fig.3.11 (Jin et al., 2010a). This piece of software is triggered every millisecond (for biological realism), though this value can be modified by a timer interrupt. This structure has been chosen to implement an abstract time paradigm, as described earlier in section 2.2.1: such an implementation, in fact, allows the various update intervals always to be kept constant. The timer interrupt is configured in software so that the interval is configurable to user specifications; these may require the ODEs to be integrated more often than each millisecond for closer biological realism (Humphries and Gurney, 2007).

The synaptic current value for each neuron is stored in a circular buffer comprising 16 slots (also known as "bins"), one for each step of the simulation. Each of these slots contains a component for excitatory current and a second for inhibitory current. The two components of the synaptic current are added to yield the current value which is injected into the neuron and are then separately processed to compute an exponential decay function, with a programmable time constant: one for the excitatory current, and a second one for the inhibitory current. The exponential decay function applied to each of the current components is:

$$I_{e|i}(t+1) = I_{e|i}(t) * (1 - \frac{1}{\tau_{e|i}})$$

where e indicates the excitatory and i indicates the inhibitory component, and $\frac{1}{\tau}$ is the decay rate. The neuron model is then integrated over the time of a simulation step

using the Euler method, and the previously computed current as input to the ordinary differential equations modelling the neuron. The equations modelling the neuron are not specified here as this is a general description of the algorithm which may apply to various neural models; the models implemented on SpiNNaker are presented in section 3.6. At the end of this computation, if the neuron is in the spiking condition, a SpiNNaker multicast network packet is generated, and the neuron is placed in the reset state. These steps are repeated for every neuron simulated by a single core.

The routing key of the multicast network packet generated represents the Global ID of the sender neuron, which is unique over all the system and comprises the fields described in Fig.3.12.

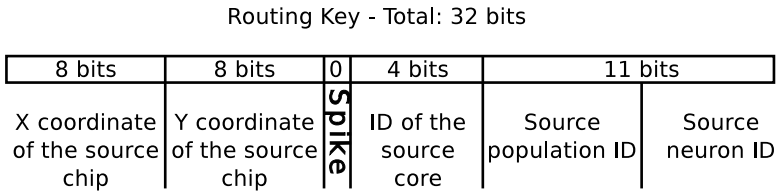


Figure 3.12: Description of the routing key structure.

The structure of the routing key is discussed in more depth in chapter 5, with the description of the routing key space assignment.

3.4.2 The incoming spike handler

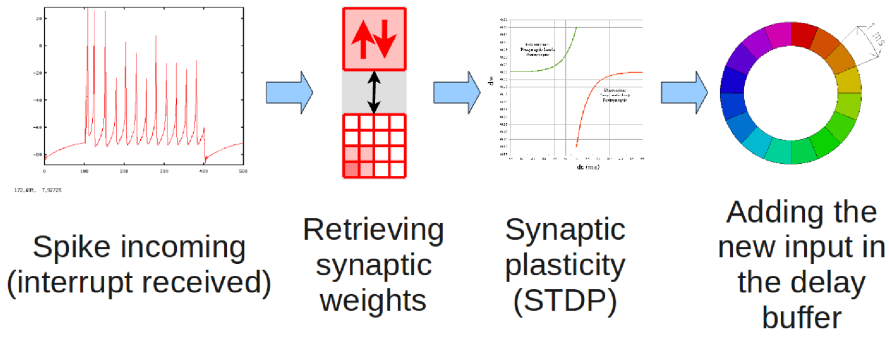


Figure 3.13: Block description of SpiNNaker incoming spike handler software

This part of the software, described in Fig.3.13, is the handler for incoming spikes, which takes care of distributing the received spikes to the correct neurons in the chip.

When a spike is received, an interrupt is generated which triggers a DMA operation to retrieve synaptic data from the external SDRAM. The synaptic words are organised

in the SDRAM by the source neuron ID, and synapses from a single source neuron form a "synaptic row": when a packet with a specific neuron ID source is received, the corresponding row is copied (by the DMA) to the TCM of the core to be processed.

The information provided by each of the synaptic words contains (all, or part of) the following information (not necessarily in the order of description):

- Synaptic delay, from 1*msec* to 16*msec*;
- Destination neuron inside the receiving core;
- Synaptic weight, generally expressed in fixed-point arithmetic format, with the number of bits representing the integer and the decimal part specific for each neuron model;
- Synaptic type: excitatory, inhibitory, etc.;
- Synaptic plasticity on/off: a single bit which specifies if the synapse is plastic or not.

Once the synaptic words are available for processing (after the DMA completes the transfer, and returns with an interrupt), the simulator applies the plasticity algorithm (if required) on the synapses, and then returns the weights to the SDRAM.

Finally, each of the weights is added to the circular buffer of the destination neuron in the bin corresponding to the synaptic delay.

3.4.3 A common point: the circular delay buffer

The SpiNNaker system is designed to deliver multicast packets from one chip of the network to all the other chips in less than a millisecond, time which is much smaller than the mean typical biological synapse.

To accommodate the typical biological synaptic delay, a circular delay buffer is used (see Fig.3.14), which is the common point between the two sections of the algorithm.

As described earlier, this buffer is an array of 16 elements ("slots" or "bins"), each containing, separately, the excitatory and inhibitory current values, whose access is determined through a pointer which indicates, always, the element relative to the current time stamp. Every interrupt received from the timer moves this pointer one element forward; when the pointer reaches the end of the array, it wraps around and points again to the first element of the array.

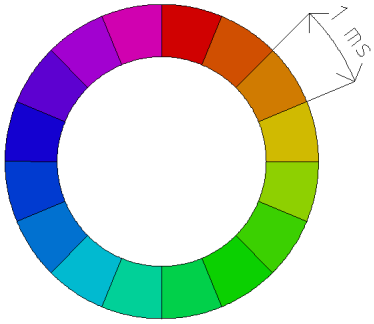


Figure 3.14: Depiction of the circular buffer used for simulating the synaptic delay

The number of slots represents the maximum synaptic delay available in terms of number of timer interrupts. If the timer interrupt is set to trigger an event every millisecond (i.e. current configuration of the software), the maximum allowed delay is 16 milliseconds. To simulate the synaptic delay, the current pointer of the buffer is displaced by the synaptic delay (wrapped around as needed) and the synaptic weight is added to the content of the correspondent bin, depending on the synapse type (excitatory or inhibitory).

3.5 Pacman

Pacman is the acronym for Partition And Configuration MANager (Galluppi et al., 2012a). This software converts a neural network description (using the standard PyNN language (Davison et al., 2008)) into binary files which need to be loaded into the SpiNNaker system to run the simulation. The block diagram of this software is depicted in Fig.3.15.

The interface "pyNN.spiNNaker" parses the description of a neural network which follows the specifications of the pyNN language. This language allows a hierarchical description of a neural network, based on populations and projections between them, allowing further hierarchically superior layers, called "assemblies", which group populations of neurons (Davison et al., 2008)).

The "pyNN.spiNNaker" interface loads the description of the neural network into a database, which holds this data, and all the further processing steps which allow the Pacman software to map it on the SpiNNaker system. In this process various constraints have to be taken into account, such as: neural model complexity, synaptic word encoding, neural parameter encoding, topology of the system in use, etc. All these constraints are stored in the "Model library" database which is used throughout

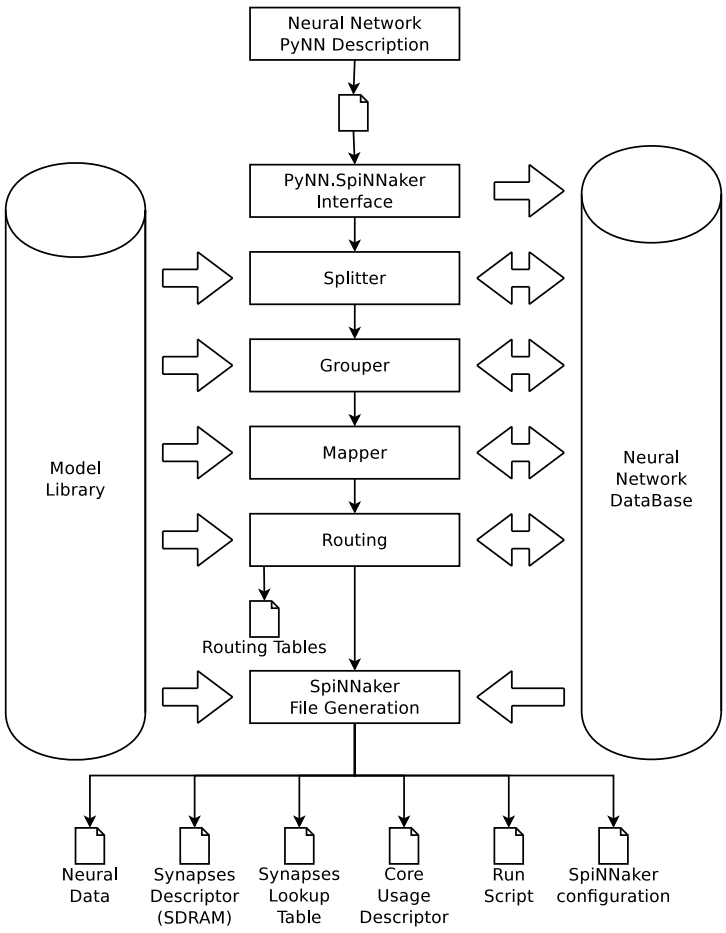


Figure 3.15: Block diagram of the partition and configuration software (PACMAN).

the network compilation process. In detail, the steps required for the network compilation process are:

1. **Splitter**: This component splits populations and projections into parts which can be mapped onto single processors. If a single population contains too many neurons to be mapped onto a single core, this population is split into multiple "proto-populations" or "part-populations", each of which can fit into a single core. This piece of software splits, also, the projections which occur between populations which are split. The split projections are called "proto-projections" or "part-projections"; the results of this step are stored back in the network database.

2. **Grouper**: This component groups populations which contain fewer neurons than those which could fit into a single core. In this way it is possible to use the SpiNNaker system for neural simulation efficiently. The outcome of this stage is again stored into the network description database.

3. **Mapper**: This component maps the outcome of the grouper on the SpiNNaker cores available, considering the topology of the network of chips and the functioning processors available.

4. **Routing**: This component assigns the addressing space to the populations of neurons and computes the routing tables for the SpiNNaker system to map the projections between the populations. This component is described thoroughly in chapter 5 of this thesis. The outcome of this process is partly written to the database, and partly written as binary files to be loaded on the SpiNNaker system.

5. **SpiNNaker file generator**: This component generates files to describe each single neuron, each single synapses and all the auxiliary files to run the simulation on the SpiNNaker system. The outcome of this process are the binary files to be loaded on SpiNNaker, or used to run the simulation. In particular, the information related to the encoding of the binary files are described, for the most part, in the model library database.

3.6 Neuron models available

Various neural models have been implemented for the software simulator. Below there is, a list of the most relevant, with the mathematical model (in terms of Ordinary Differential Equation(s) - ODE) that each of these implement:

3.6.1 Izhikevich neuron

A model of the Izhikevich neuron (Izhikevich, 2003) has been implemented for the SpiNNaker software simulator (Jin et al., 2008). The ODEs representing this mathematical model are:

$$\begin{cases} \dot{v} = 0.04v^2 + 5v + 140 - u - I \\ \dot{u} = a(b \cdot v - u) \end{cases}$$

$$\text{if } v = 30mV \text{ then } v = c,\ u = u + d$$

Where the state variable v is the neuron membrane potential, while the state variable u is the neuron recovery variable. The condition $v = 30mV$ is the spiking condition. Whenever this condition is met, the neuron "fires" (emits an action potential) and

then goes back to the reset state expressed in the same formula.

The constants 0.04, 5 and 140 have been chosen by Izhikevich (2004); using these it is possible to simulate at least 20 biologically meaningful spiking behaviours. a, b, c and d are the four parameters of the Izhikevich neuron model (Izhikevich, 2003):

- The parameter a describes the time-scale of the recovery variable u. Smaller values result in slower recovery;
- The parameter b describes the sensitivity of the recovery variable u to the sub-threshold fluctuations of the membrane potential v. Greater values couple v and u more strongly resulting in possible sub-threshold oscillations and low-threshold spiking dynamics;
- The parameter c describes the after-spike reset value of the membrane potential v caused by the fast high-threshold K^+ conductances;
- The parameter d describes after-spike reset of the recovery variable u caused by slow high-threshold Na^+ and K^+ conductances.

The mathematical equations are integrated twice in each time step to avoid the mathematical instabilities suggested by Izhikevich (2010): a biologically unrealistic oscillation of the membrane potential due to an integration step too long.

3.6.2 Leaky Integrate-and-Fire neuron

A model of the Leaky Integrate-and-Fire neuron (Dayan and Abbott, 2001) has also been implemented for the SpiNNaker system (Rast et al., 2010b). The mathematical model of this neuron is:

$$\tau \frac{dV}{dt} = V_L - V + R_m \cdot I_{input}$$

The spike condition is defined by the condition $V \geq V_{threshold}$. Whenever the neuron membrane potential satisfies this condition, it emits a spike and then the membrane potential resets to V_{reset}.

Where:

- V is the neuron membrane potential;
- τ is the membrane time constant of the neuron. If the input is null, the membrane potential exponentially relaxes with this time constant;

- V_L is the resting potential of the cell. If the input is null, the membrane potential of the neuron decays exponentially to this value;
- R_m is the membrane resistance;
- I_{input} is the current injected in the neuron;
- $V_{threshold}$ is the threshold potential for a spike emission;
- V_{reset} is the reset value for the membrane potential after the neuron emits a spike;

This neuron model has been implemented in various versions according to the precision required for the simulation: the first implementation used 16 bits in fixed point precision, with 8 bits representing the integer part and 8 bits representing the decimal part.

A second implementation has been produced with 32 bit precision: 16 bits to represent each of the integer and decimal parts. This implementation replaced the 16 bit one.

3.6.3 Poisson spike source generator neuron

This neuron generates a spike train according to a Poisson process: the Inter-Spike Interval (ISI) is an instance of a Poisson random variable generated during the simulation.

Limitations in the implementation, due to the fixed-point arithmetic, limit the rate of the spike train in the interval between $25Hz \leq rate \leq 1000Hz$.

3.6.4 Spike source neuron

This neuron generates spikes according to a pattern saved in the SDRAM memory chip. Due to limitations in the memory available for this purpose, the input pattern has to be less than $8MB$.

3.6.5 Spike source live neuron

This neuron generates spikes according to data input from a host computer through the Ethernet channel. This neuron works in partnership with the SpikeServer software that will be introduced in the next chapter.

3.6.6 NEF interface neurons

To import the Neural Engineering Framework (NEF) (Eliasmith and Anderson, 2004) into the SpiNNaker simulator, a value encoder neuron and a value decoder neuron are required (Galluppi et al., 2012b), as described earlier in section 2.3.2

3.7 Plasticity models available

In the SpiNNaker architecture synaptic weights are available for computation only when a spike event is received. Therefore, to trigger potentiation when the post-synaptic neuron emits an action potential, an implementation model is required to store this information until the subsequent pre-synaptic spike is received. This model is called the "Deferred Event Model" (Rast et al., 2009), and has been applied to reproduce two learning rules. The first algorithm has been developed according to the Spike-Timing-Dependent Plasticity (STDP) general rule (Bi and Poo, 1998; Jin et al., 2009). A second algorithm has been developed to reproduce a simplified learning rule: the spike-pair STDP (also known as nearest-neighbour STDP).

However, these two learning rules presented high requirements in terms of computational complexity (see section 6.12.1 for details), which, in some cases, override the real-time constraint of the simulation. To overcome this negative aspect, this dissertation presents, in chapter 6, a third learning rule, the STDP-TTS, which represents the author's main research contribution.

3.8 Summary

This chapter presented the SpiNNaker project, starting from the hardware, to the neural network simulation software. According to the classification scheme proposed earlier in chapter 2, the SpiNNaker simulator can be classified as a digital hardware, real-time, neural network simulator which allows multiple neuron models to be used during a single simulation; it implements learning features and runs with an abstract-time paradigm. The comparison table 2.1 presented the features described in this chapter.

The topics presented here form the background on which the author's research has been carried out. Starting from the next chapter the author's three contributions are presented.

Chapter 4

SpikeServer

4.1 Introduction

This chapter presents a novel software PLL (Phase Lock Loop) to synchronize two processes resident on different computational units which exist in diverse clock domains. This method has been successfully applied to an experimental real-time neuromorphic architecture, where synchronised external temporal stimuli are provided to neurons being simulated inside the system. The implementation and results of this software are provided as a case-study of how the technique may be employed to link together two incoherently clocked domains. The chapter finally presents a study of the efficacy of the technique when applied across further systems and deployment scenarios, focusing on the time-drift and applied correction characteristics between the two computational units.

The techniques presented in this chapter, and the next, were essential to perform the required simulations to test the learning rule which will be presented in chapter 6. In fact, the only other options to inject a well-defined set of spikes into the system (which was developed before the SpikeServer) is the SpikeSource neural population (see section 3.6.4), which has a limitation in the number of spikes that may be loaded into the system. Therefore for long simulations, with the number of neurons used for testing the learning rule (800 input neurons), the development of this technique was required. This software was developed to allow long simulations with the number of neurons not restricted by the memory available and, in addition, to solve the wider problem of how to interface spiking neurons simulated in the SpiNNaker architecture with the external world.

4.2 Real-time systems background and introduction

Real-time systems are specialised platforms which guarantee, to a greater or lesser extent, the execution of a task in a particular time-frame. Such systems have to remain reliable within their particular bounds, and fall generically into two categories: hard and soft real-time systems.

Hard real-time systems are those where a failure subsequent to a miss in the timing constraint is considered a fatal fault (e.g. aircraft control, pacemaker device, car injection system, etc.).

Soft real-time systems are those in which a failure subsequent to a miss in a timing constraint is undesirable, and may generate some system degradation, but the system itself can continue its job (e.g. display screen update, audio/video streaming, etc.).

Approaching these constraints from a design perspective, it is a hard real-time system if a formal validation of the timing is required (Liu, 2000). A soft real-time system, on the other hand, does not require such a formal validation, but statistical constraints suffice for bounding purposes.

Within the field of spiking neural network simulators (Maass, 1997), real-time systems are commonly used to maintain the temporal integrity of simulations. This is the case, here, as the arrival-time of spikes is commonly believed to be an encoding technique for information passing in the brain. Neural network simulators operate using an electronic substrate and simulate the biological 'wetware' in which mathematical models provide the behaviours for neurons, synapses and learning within the simulation (Misra and Saha, 2010; Ponulak and Kasinski, 2011).

Spiking Neural Network (SNN) simulators have a range of real-time requirements. A hard requirement, for example, is the neuron membrane potential update: if an update does not meet its deadline, the result of the whole simulation is incorrect. An example of a soft real-time requirement is given by the synaptic current injection: if the input to a particular neuron is not provided in the right time-frame, the result of the simulation is degraded but not necessarily invalid. This latter example matches with biological observations where the synaptic delay may vary (Bakkum et al., 2008).

When operating SNN simulations, input stimuli may be provided from many sources, including spikes from afferent neurons, and from interfaces to external environments. Devices which mimic the senses are high on the agenda, after all, an artificial neural network is simulating the biological brain. Examples of input sensors which can offer 'brain-like' spiking output include silicon retinas (Lichtsteiner et al., 2008; Leñero-Bardallo et al., 2010) and silicon cochleas (van Schaik and Liu, 2005).

The configuration of the sensors, and the network design itself all contribute to the output(s) of the SNN. In biology motor functions are one common output product and, in the simulation domain, these can be replicated by actuating a motor controller for example (Davies et al., 2010).

Each neural network project and simulator has unique targets with specific traits and characteristics. There has been some work in the SNN field to define a high-level standard representation of events in a neural network, resulting in the Address Event Representation protocol (Boahen, 2000). However, input and the output systems are generally tailored to match the requirements of the target system; it is possible to identify two distinct interfacing families:

1. **Analogue input/output** for analogue hardware simulators, such as FACETS (Schemmel et al., 2010) or VLSI chips (Livi and Indiveri, 2009);

2. **Digital input/output** for hardware simulators (either analogue or digital) which implements digital communication systems between neurons, such as Neurogrid (Merolla et al., 2007) or FPGA simulators (Cox and Blanz, 1992). AER (Boahen, 2000) is often used for identification and timing means here.

This chapter presents a novel software architecture, the SpikeServer, which was initially developed to ensure that the timing of external input stimuli meets the requirements of a SpiNNaker simulated neural network, and this is used as the initial case-study. However, the approach taken may be useful in areas broader than just the neural computing space – wherever there is a requirement to synchronise the operation of two or more arbitrary computational units over different clock domains, using a non-deterministic interconnection channel. The basis of the SpikeServer is formed by a novel general-purpose software PLL, which is presented in this chapter.

4.3 Architecture of the SpiNNaker system

As described in chapter 3, the SpiNNaker simulator comprises a hardware computational substrate on top of which a neural network software simulator has been developed. In this chapter one of the peculiarities of the SpiNNaker architecture is analysed in more in-depth: the asynchronicity of the SpiNNaker system.

4.3.1 Asynchronicity of the system

The SpiNNaker system contains ARM cores and memory banks, which are synchronous components, however its communications operate asynchronously. Each chip, therefore, runs on the basis of its own independent clock, islands of synchrony in a sea of asynchronous interconnection. This configuration is known as Globally Asynchronous, Locally Synchronous (GALS) (Chapiro, 1984), and offers good energy efficiency without the requirement for a clock to be distributed globally throughout the system. However, this configuration has a drawback: when regarding the system from an external perspective (such as a host computer connected to the SpiNNaker system), there is no signal which identifies the clock phase of the machine.

Despite the global asynchronous nature, when injecting spikes into the system to stimulate its neurons, the timing must be matched to that of the target core to attain consistent simulation. To deal with the asynchronous nature of the SpiNNaker system, a high-level software PLL synchronization mechanism has been designed and incorporated in the external SpikeServer; this is fully described in the next section.

4.4 Implementation of SpikeServer

The SpikeServer is an interrupt-driven real-time multi-threaded software program that runs on a host computer and which interfaces and synchronizes with the SpiNNaker system, receiving and sending spikes at the appropriate time. The main structure of this software is depicted in Fig.4.1.

The software is composed by more than $1,200$ lines of C++ code (summarized using pseudo-code for clarity) and spawns three threads which are put, initially, in the "wait" state. Each of these threads is triggered by particular events:

1. **Spike input parser**: this thread parses the input spikes and queues them into a buffer, ready to be sent to the SpiNNaker system at the appropriate time. The thread starts when a new spike to be queued is received and needs to be sent to the SpiNNaker board; This thread is described with the pseudo-code of the Algorithm 2.

2. **SpiNNaker packet sender**: this thread sends spikes from the host computer to the SpiNNaker system through the Ethernet channel according to the timing constraints; this thread is activated by a timer synchronised with the target core on the SpiNNaker system. Since the SpiNNaker system updates neuron state

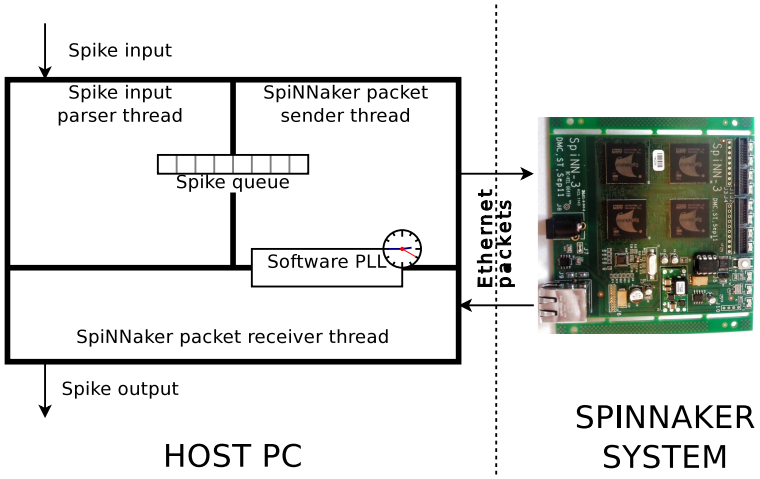


Figure 4.1: Schematic diagram of the SpikeServer software.

variables once per millisecond, the input must be provided at this frequency too; however, to keep the timing constraint within this boundary, and effectively detect and adjust for timing drifts, a period of 500μsec is used to increment the internal counter. Pseudo-code of the Algorithm 3 summarizes the functionalities implemented for this thread.

3. **SpiNNaker packet receiver**: this thread receives packets from the SpiNNaker system and deals with them according to their type; it is activated by an Ethernet packet received interrupt, and the pseudo-code of Algorithm 4 described its behaviour.

The two threads "Spike input parser" and "SpiNNaker packet sender" form a producer-consumer paradigm using a shared queue which stores the spikes to be sent. The synchronization routine performs a software PLL function which is the equivalent of a hardware PLL: it receives packets from the SpiNNaker system with a known inter-packet time, and from that infers the phase and the period of the SpiNNaker system timer tick. Using this information, the software is then able to send spikes using a locally derived clock which is synchronized with that of the target SpiNNaker chip by the software PLL. As the host PC has a clock which may drift in time relative to that of the SpiNNaker clock, it adjusts to resynchronization packets from SpiNNaker which is considered the master. The SpikeServer software is now able to send spikes according to a clock dynamically synchronized with the SpiNNaker board.

Algorithm Name: SPIKE INPUT PARSER

Description: This thread reads data from an input stream and checks if it identifies the end of a millisecond frame (0xFFFFFFFF): if yes, the thread is stopped until the next timer interrupt, otherwise the received data is considered a spike and queued in a list to be sent to the SpiNNaker board. The condition of a timer interrupt arriving before the end-of-frame identifier (0xFFFFFFFF) is considered, and in this case the spikes received until the subsequent end-of-frame are dropped.

```
while (data_in_the_input_stream()) do
    data = read_one_word_from_stream();
    if (spike_queue_not_in_use()) then
        if (data == 0xFFFFFFFF) then
            thread_wait_for_timer_signal();
        else
            queue_spike(data);
        end
    else
        while (data_in_the_input_stream() && (data != 0xFFFFFFFF)) do
            drop_spike(data);
        end
    end
end
```

Function description:

data_in_the_input_stream(): checks if data from the input stream is available;
read_one_word_from_stream(): reads 32 bits from the input stream;
spike_queue_not_in_use(): the spike queue forms a shared memory area between a producer thread (this thread) and a consumer threads (SpiNNaker packet sender - described later); therefore the access to this area needs to be arbitrated;
thread_wait_for_timer_signal(): puts the thread in wait state for a signal;
queue_spike(): puts the spike just read from the input stream in the producer-consumer shared memory;
drop_spike(): drops the data just received without further processing;

Algorithm 2: Parser of the spikes to be sent to the SpiNNaker board.

Algorithm Name: SPINNAKER PACKET SENDER

Description: This thread reads, from the shared memory, the spikes already packed to be sent to the SpiNNaker board and sends them according to a timer interrupt event. This thread is the consumer of the data produced by the thread "Spike input parser".

---

if *(producer_thread_not_in_wait_state())* **then**
  | *signal_timer_trigger_to_producer()*;
end
if *(spike_queue_in_use())* **then**
  | *thread_wait_for_queue_free()*;
end
spikes = *flush_spikes_from_shared_memory()*;
send_to_SpiNNaker(spikes);
free_memory(spikes);
signal_producer_thread();

Function description:

producer_thread_not_in_wait_state(): checks whether the producer thread ("Spike input parser") is in wait state meaning that has completed loading the spikes for the current time frame;

signal_timer_trigger_to_producer(): signals to the producer thread that the timer signal has been triggered, and therefore no more spikes should be loaded in the shared memory for the current time frame;

spike_queue_in_use(): checks if the shared memory is currently being accessed by the producer thread;

thread_wait_for_queue_free(): the thread is put in wait state until the shared memory becomes available;

flush_spikes_from_shared_memory(): retrieves the spikes that have been queued in the shared memory, freeing it for the producer to restart;

send_to_SpiNNaker(): sends the spikes to the SpiNNaker system;

free_memory(): frees the memory used by the spikes that have already been sent to the SpiNNaker system;

signal_producer_thread(): signals to the producer thread that the queue is ready to accept the data for the following time frame.

Algorithm 3: Sender of the spikes to the SpiNNaker system.

Algorithm Name: SPINNAKER PACKET RECEIVER
Description: This thread receives Ethernet packet from SpiNNaker; on the basis of the content, each packet is identified as a resynchronization packet or a spike packet, and addressed correspondingly.

---

if *(data_received_is_sync_packet())* **then**
| *resynchronize()*;
end
if *(data_received_is_spike_packet())* **then**
| *forward_to_output_stream()*;
end

Function description:
data_received_is_sync_packet(): checks whether the packet received from the SpiNNaker board contains synchronization data;
resynchronize(): triggers the software PLL routine to resynchronize the host clock with the SpiNNaker one;
data_received_is_spike_packet(): checks whether the packet received from the SpiNNaker board contains spike data;
forward_to_output_stream(): forwards the spikes received to the output stream;

Algorithm 4: Data receiver from the SpiNNaker system.

The packets received from the SpiNNaker system may belong to one of two possible classes: spike packets, which are opened and forwarded to the output channel, or synchronization packets, sent once every 256*msec*, used internally within SpikeServer for derivation of the time in the SpiNNaker machine, and consequently to adjust the SpikeServer internal timer. The behaviour of the routine that performs the required adjustments is described in Fig.4.2, and with the pseudo-code of the Algorithm 5, and consists of three main phases:

- **Phase 1**: The SpikeServer runs using a command line parameter which initiates the timer period with the required configuration;
- **Phase 2**: When the first synchronization packet is received from the SpiNNaker board, the software locks to the phase triggering the timer event immediately, leaving the timer period unchanged. The synchronization packet contains the

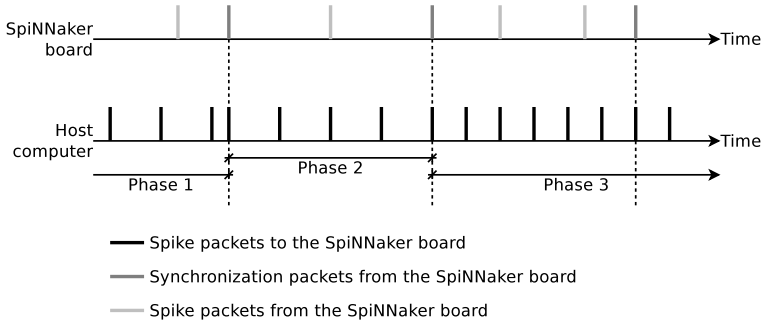


Figure 4.2: Diagram of the phases of the software PLL (see the description in the text).

time stamp of the SpiNNaker system when it was sent, which information is stored by the SpikeServer;

- **Phase 3**: When the second synchronization packet is received, the software is able to make a comparison between the SpiNNaker time which has passed between its synchronization packets, and the time which has passed for the host computer. Due to jitter in handling interrupt requests in both systems, variances in the crystals which generate the base frequencies, and delays in the delivery of network packets, the two clocks usually have different periods. In this third phase the timer event is triggered immediately and the timer period adjusted so that the period for the host CPU matches the period of the SpiNNaker board.

Subsequent synchronization packets are used to maintain the synchronization between the SpiNNaker board and the host computer, and to adapt to dynamic variations (e.g. temperature induced drifting).

Algorithm Name: TIMER RESYNCHRONIZATION ROUTINE

Description: This routine performs the PLL operation resynchronizing the host with the SpiNNaker time clock. It is performed on the basis of the network packets received from the SpiNNaker system, with particular regards to the "*synchronization packets*". Using the first of these, the phase of the clock is adjusted. Starting from the second packet both phase and period are adjusted. In particular, the period is adjusted using the host time interval as a reference.

---

if *(first_resync_packet())* **then**
    store_current_host_timestamp();
    store_SpiNNaker_timestamp();
    recharge_timer_period();
    trigger_timer_in_50_µs();
else
    host_time_interval = current_host_timestamp - last_stored_host_timestamp;
    SpiNNaker_interval = current_SpiNNaker_timestamp -
    last_stored_SpiNNaker_timestamp;
    set_new_timer_period (host_time_interval / SpiNNaker_interval);
    if *(current_SpiNNaker_timestamp* $\geq$ *current_host_timestamp)* **then**
        trigger_timer_in_one_timer_period();
    else
        trigger_timer_in_50_µs();
    end
end

Function description:

first_resync_packet(): checks whether the sync packet received is the first;
store_current_host_timestamp() and *store_SpiNNaker_timestamp()*: stores in memory the correspondent timestamp for future use;
recharge_timer_period(): reloads the timer period with the value available from the user;
set_new_timer_period (): computes the new timer period from the available data and sets the timer period accordingly;
trigger_timer_in_50_µs(): sets the timer to trigger an event in 50 μs;
trigger_timer_in_one_timer_period(): sets the timer to trigger an event in a full timer period;

Algorithm 5: Timer resynchronization routine.

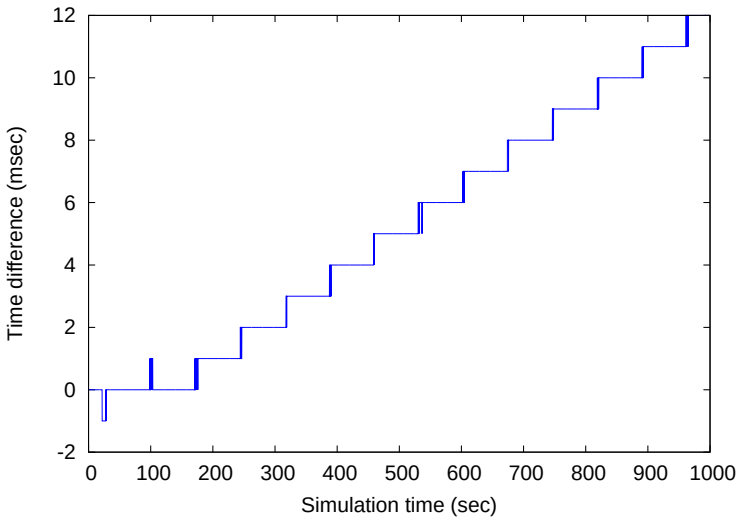


Figure 4.3: Depiction of time drift between the host and the SpiNNaker board. No synchronization routine is used in this experiment.

4.5 Tests and results

To illustrate the function of the software PLL introduced in this chapter, the results of the execution of the SpikeServer without any synchronization mechanism is presented; this highlights the drift which would otherwise occur in the time reference between the source and the destination computational substrates. The software PLL resynchronization routine is then added to show the improvements. This experiment is performed using a SpiNNaker board connected to a laptop running a standard release of the GNU/Linux Ubuntu operating system. To minimise the impact of the operating system and network overheads the following three constraints are applied:

1. No graphical interface running at the time of the experiment;
2. Network mapping entries (ARP) are set manually before the experiment starts;
3. The packet sender and the packet receiver are directly connected through an Ethernet cable.

The three constraints are required to keep the time required to handle the interrupts coming from the various sources as low as possible. In practice the graphical interface

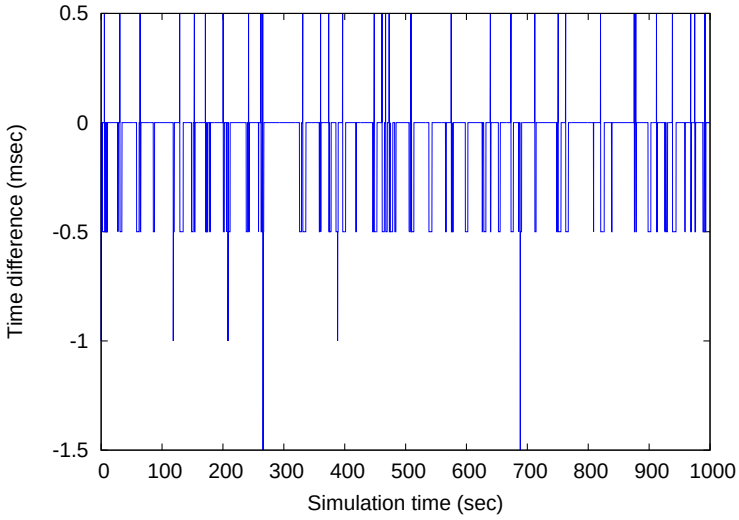


Figure 4.4: Depiction of time drift between the host and the SpiNNaker board, with the software PLL synchronization technique applied.

was found to introduce a variable latency of many milliseconds before the timer interrupt was processed. At the same time, it is useful to avoid ARP requests which may take longer than a millisecond to complete, and, for the same reason, external network equipment which may increase the delay and the jitter in the delivery of the network data is removed.

The SpiNNaker board sends network packets to the host PC with its internal time reference. The receiver PC compares this time against its internal reference, calculating the uncorrected time drift. Fig.4.3 shows the result of this test: the simulation time (in seconds) is on the horizontal axis, and the vertical axis displays the cumulative drift (in msec) for the host PC since the first received packet. In Fig.4.3 it is possible to discern a linear pattern every $\approx 70sec$ where the difference between the SpikeServer time and the SpiNNaker board (used as reference) increases by $1msec$, implying that the clock on the SpikeServer system runs faster than the time reference in the SpiNNaker board (by $\approx 0.0015\%$).

A second experiment shows the behaviour *with* the resynchronization routine and the results are plotted in Fig.4.4. In this case the error is generally within the tolerated range $\pm 500\mu sec$. Where the SpikeServer is faster than the SpiNNaker system (error of $+500\mu sec$) the SpikeServer waits one $500\mu sec$ time slot before sending the network packet containing the set of spikes to the SpiNNaker board. However, given the slow drifts between clocks typically experienced (e.g. Fig.4.3), the spikes are still likely

to arrive at the SpiNNaker board in the right time slot. In the second case, when the SpikeServer is slower (error of $-500\mu sec$), the spikes may be sent half a millisecond later than the beginning of the SpiNNaker millisecond slot; however, even considering the network delay, they reach the SpiNNaker system within the remaining $500\mu sec$ of the time slot, therefore avoiding errors in the neural simulation. Only in five cases was the error larger than or equal to $-1msec$. Since, in each of these intervals, it is not possible to discriminate when the drift occurred, to obtain a comparison value with the previous error rate, the drift is considered to occur at the very beginning of the interval (worst case scenario), and therefore the error is $2.5msec$ over the length of the test, which was $1,000,000msec$. Therefore, the error lasted, in these conditions, for $2.5/1,000,000 \approx 0.00025\%$ of the time of the experiment, with an accuracy increased by one order of magnitude. However, these errors are always temporary and are corrected at the next occurrence of the synchronization packet when the SpikeServer resynchronizes to the SpiNNaker's internal clock reference.

4.6 Discussion

This chapter introduced an implementation of a software PLL used to synchronize two computational substrates through an Ethernet link – a non-real-time channel. This software demonstrated the ability to keep within the timing boundaries for all but $5/1,000,000 = 0.0005\%$ of cases. These errors may be attributed to the different periods of the clock references in the two computational substrates, and to the non-deterministic interrupt handling routines (particularly on the host computer). Three constraints have been introduced in the implementation of such an architecture on a standard GNU/Linux machine (determined through experiment) described earlier in the chapter. Experimentation proved that when using an overlying graphical interface the interrupts may be delayed by up to $5msec$ which significantly disturbs synchronization. Similarly, the resolution of the dynamic IP to MAC address ARP mapping requires more than $1msec$ to complete, which again affects the results of the simulation whilst being performed. The overall goal of the constraints listed is to reduce the number of processes which can block the timing accuracy of the interrupt handlers used in the SpikeServer. Equally, to minimize the delay and jitter in the network, all the networking equipment which may add timing inconsistencies to the communications traffic has been removed.

This software PLL technique has been applied to a specific case study of the SpiNNaker neuromorphic hardware. However, without any particular modification, it may be applied to synchronize diverse platforms for any desired purpose.

4.7 Summary

This chapter presented a novel technique used to synchronize two computational units across a non-real-time channel (like the Ethernet channel). The next chapter presents a novel technique used to route spikes between populations of neurons, instead of the usual neuron-to-neuron spike routing used in the simulators so far described. The two techniques have been applied to test the learning rule presented in the chapter 6, which is the main core on this research work.

Chapter 5

Population-based routing

5.1 Introduction

SpiNNaker is a hardware-based massively-parallel real-time universal neural network simulator designed to simulate large-scale spiking neural networks. Spikes are distributed across the system using a multicast packet router. Each packet represents an event (spike) generated by a neuron. On the basis of the source of the spike (chip, core and neuron), the routers distribute the network packet across the system towards the destination neuron(s). This chapter describes a novel approach to the projection routing problem which shows advantages in both the size of the routing tables generated and the computational complexity for the generation of routing tables. To achieve this, spikes are routed on the basis of the source population, leaving to the destination core the duty to propagate the received spike to the appropriate neuron(s). The features presented are required for a faster reconfiguration of the SpiNNaker system, even if they do not affect the learning behaviour of the system, to allow the overall system to deliver the promise of reaching $\approx 1,000,000,000$ neuron simulation, which is $\approx 1\%$ of the human brain neuron count.

5.2 Background

Biological neurons communicate in networks with spectacular levels of connectivity using signals which are remarkably simple. The structure of the input dendritic tree allows a neuron to receive inputs from many thousands of other neurons through synaptic connections. The nature of the network – simple processing, complex connectivity – suggests that much of the remarkable processing power of the brain must stem from

the connection topology.

The need to support efficient processing within the wiring constraints and the space of the brain naturally suggests some internal structure: biological neural networks do not have random topologies. While specific functional microcircuits within biological neural networks have been identified in some cases (e.g. Lefort et al. (2009)), the global connectivity of the brain is known to feature bundles of long-range connections linking specific regions to other specific regions in tight clusters (Hagmann et al., 2008). Likewise, the general structure of the cortex has been analysed and is understood to have dense local connectivity with relatively sparse, but strongly-directed, long-range connectivity (Binzegger et al., 2009). The overall pattern – heavily clustered populations of neurons communicating through narrow specific projections – is thought to provide a generic modular architecture to the brain which permits efficient processing of almost any function or behaviour within a universal substrate.

It seems remarkable, therefore, that attempts to create neural networks in hardware have rarely emphasised or even implemented this type of architecture. First-generation neural processors, on the whole, tended to implement either cellular connectivity or all-to-all patterns that promote reduced locality (Lindsey and Lindblad, 1995). A second generation, acknowledging the need for reconfigurability, adopted a rewirable architecture, but on the whole tended towards the circuit-switched approach typical of FPGAs which is unsuitable for biological-style topologies (Maguire et al., 2007). Some examples of neuron intercommunication paradigms, as implemented in hardware spiking neural network simulators (Draghici, 2000), are:

- **Fully connected networks**: neurons communicate with all or a subset of the remaining neurons;
- **Locally connected networks**: neurons are arranged in a matrix in which each neuron communicates only with a subset of the neighbouring neurons;
- **Layered architectures**: neurons of a specific layer communicate only with neurons of the layers up-stream and down-stream;
- **Reconfigurable architectures**: the connections of each neuron are reconfigurable both in synaptic destination and in synaptic weight.

A common feature of all these architectures is that only *neuron-to-neuron* connections are taken into account, i.e. the description of the connectivity always uses a single neuron as source and a single neuron as destination. However, in biology, it is known

that neurons are functionally aggregated into populations (called "cell assemblies" in Hebb (1949)) where all the neurons contribute to the same function, and then populations are interconnected (Fingelkurts et al., 2005). Indeed, in biology, the per-unit computation seems to be very simple while the sophistication is in the connectivity. It is surprising, therefore, that over 2 decades of hardware (and software) development, effort has focussed more on efficient *computational methods* rather than on efficient *connectivity architectures*. The idea of improving the connectivity architecture seems always to have been latent in the field (James and Hoang, 1992), but never exploited.

In this chapter a software approach used to configure the SpiNNaker architecture will be discussed. This software is based on the novel idea of connecting *populations*, where all neurons in a population share a common path to the destination population(s). The precise description of the connection between each single neuron happens only at the very last operation, when the spike has to be distributed within the destination population which resides in a specific simulation unit. The population-based approach is then tested with various models of neural networks and shown to be an efficient approach to the neuron inter-communication problem.

Given the high fan-out of neural connectivity, generating the distribution trees and filling the spike routing tables is a very important, *non-trivial* part of the neural computation: in this chapter a new approach which simplifies this task is explored. Instead of generating neuron-to-neuron multicast trees the computational time is greatly reduced by generating population-to-population routes.

5.3 Architecture of the SpiNNaker system

The architecture of the SpiNNaker system (as highlighted in chapter 3) has been designed for neural network simulation, even though it not limited to this task. However, for the scope of this chapter, it is important to focus on the propagation of neural action potentials, and how it has been designed in the SpiNNaker architecture.

5.3.1 Multicast communications in SpiNNaker

When a neuron fires, a multicast packet is generated and transmitted through the network; the only information contained in the packet is the identifier of the neuron which

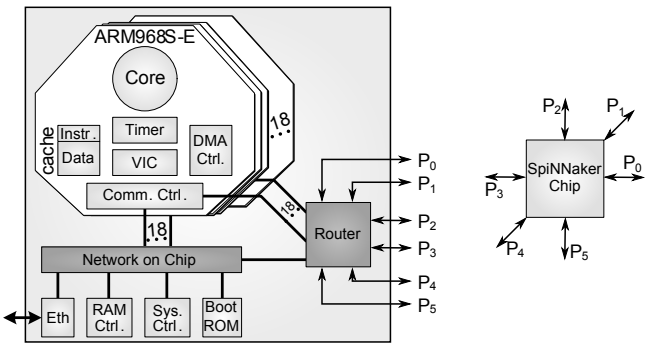


Figure 5.1: Schematic of the SpiNNaker chip with the view of the routing directions.

has fired (see Fig.5.2). The information necessary to deliver the packet to the appropriate destinations is embedded in the routing table present in the router of each SpiNNaker chip (Plana et al., 2007). The mechanism through which network packets are routed has been described in chapter 3.

In the following sections of this chapter the term "hop" is used to describe the passage of a projection (or, equivalently, a packet) through one router of a SpiNNaker chip.

5.4 Population-based routing approach

In the SpiNNaker system, each action potential generated by a neuron is sent to destination core(s) using a multicast packet on the basis of the source neuron identifier. The routing key comprises 32 bits uniquely identifying the source neuron which emitted the action potential.

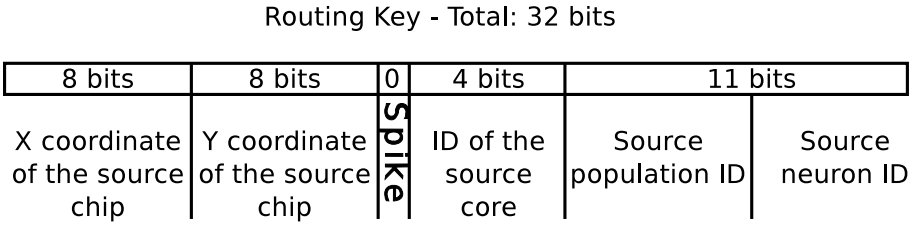


Figure 5.2: Description of the routing key structure.

The 32 bits forming a routing key can be used arbitrarily, since the routers and cores are fully programmable. However, for the sake of simplicity, a convenient structure has been defined with the five fields described in Fig.5.2. The first two fields (16

bits) provide information about coordinates (X and Y) of the chip where the neuron is placed. The third field, 1 bit, is used to distinguish between a spike (bit set to 0) and a system packet (bit set to 1); the fourth field identifies the source core inside the chip where the neuron is placed. The last field is further split into two parts: a *population* identifier and a *neuron* identifier. which have varying sizes so that it is possible to adapt them to the size of the populations simulated in each core. If populations are large, there can be fewer populations inside a core (few bits for the population identifier, many bits for the neuron identifier). However, inside each core there can be a greater number of smaller populations (many bits for the population identifier, few bits for the neuron identifier). If populations exceed the number of neurons which a core is able to simulate, they can be split into appropriately sized parts which will be treated as individual populations, returning the problem to the original population-based routing.

The masking capability implemented in the router allows selection of the bits which determine how the packet has to be routed. This feature allows selection of the part of the routing key related to the first five fields depicted in Fig.5.2, which represent the complete source population identifier.

For the purpose of the generation of the routing key, populations are sorted inside each core by size; additionally, the size of the population is rounded up to the nearest power of two. This addressing scheme has the advantage of subdividing the routing key so that all neurons belonging to the same population will have the same initial pattern, which is the population key identifier code (population ID). However, there is the disadvantage that some of the routing keys are not actually used for specific neurons, since they are used to align the following routing keys so that each population has its own identifier inside a core.

An example of this addressing scheme is represented in Tab.5.1 where the first column represents the population size. The "Rounded size" column represents the population size rounded up to the nearest power of two. The following two columns represent how the neuron identifier field (or equivalently the routing key) is split between the population identifier and the neuron identifier in terms of number of bits. The last column represents the number of routing keys which are unused: the difference between the real size of the population (first column) and the number of routing keys assigned to the population (second column).

The greatest number of unused routing keys is generated if a population contains $2^n + 1$ neurons: the size of the neuron identifier is then $n + 1$ bits, (2^{n+1} possible routing keys) of which $2^n + 1$ are actually used and $2^n - 1$ are unused, with a ratio of

Population size	Population Rounded size	Population ID size (bits)	Neuron ID size (bits)	Unused routing keys
1	1	11	0	0
2	2	10	1	0
3	4	9	2	1
...	...	...	...	...
31	32	6	5	1
32	32	6	5	0
33	64	5	6	31
...	...	...	...	...
63	64	5	6	1
64	64	5	6	0
65	128	4	7	63
...	...	...	...	...
999	1024	1	10	25
1000	1024	1	10	24

Table 5.1: Set of routing keys used and unused for each size of the neural population.

$\approx 50\%$. Therefore, to include this worst case scenario in the addressing scheme, 11 bits of the routing key are assigned to the population and neuron identifier. In total this addressing space is able to identify $2,048$ neurons per core in the SpiNNaker system, giving twice the addressing space strictly required for the simulation. The number of neurons for which the current SpiNNaker system has been designed is $1,000$ per core (maximum); this value strongly depends on the complexity of the model, however, and on the interconnection and activity pattern.

The key point of the approach to population-based routing is the definition of the population key; in the following two examples, the first part of the routing key, which includes the X and Y coordinates of the chip and the core identifier, are not specified. The examples focus only on the section related to the population and neuron IDs, describing how these values are computed. In a simple example, if a core contains a single population of 60 neurons, the population will be assigned the routing keys in the set 00000000000 to 00000111111, even though the last four routing keys will not be used. Therefore the population ID assigned to this population comprises 5 bits 00000, which is the immutable part of the set of the assigned routing keys:

$$\overbrace{00000}^{\text{Population ID}}\ \overbrace{xxxxxx}^{\text{Neuron IDs}}$$

In a more complex example, if a core contains three populations of (A) 60, (B) 20

and (C) 6 neurons, the routing keys will be assigned consecutively (see Tab.5.2).

Population Name	Population size	Routing key start	Routing key end	Population ID	Neuron IDs
A	60	00000000000	00000111111	00000	xxxxxx
B	20	00001000000	00001011111	000010	xxxxx
C	6	00001100000	00001100111	00001100	xxx

Table 5.2: Example of routing for three neural populations.

For the purpose of routing the packets in the SpiNNaker network, it is possible to use a single entry in the router per population, defining the route on the basis of the population ID. Since this has a variable size, the mask field of each routing entry must be computed according to the population size (rounded up to the greater power of two). In this way, even if populations are formed by a number of neurons which is not exactly a power of two, it is still possible to use a single routing entry per population to route packets.

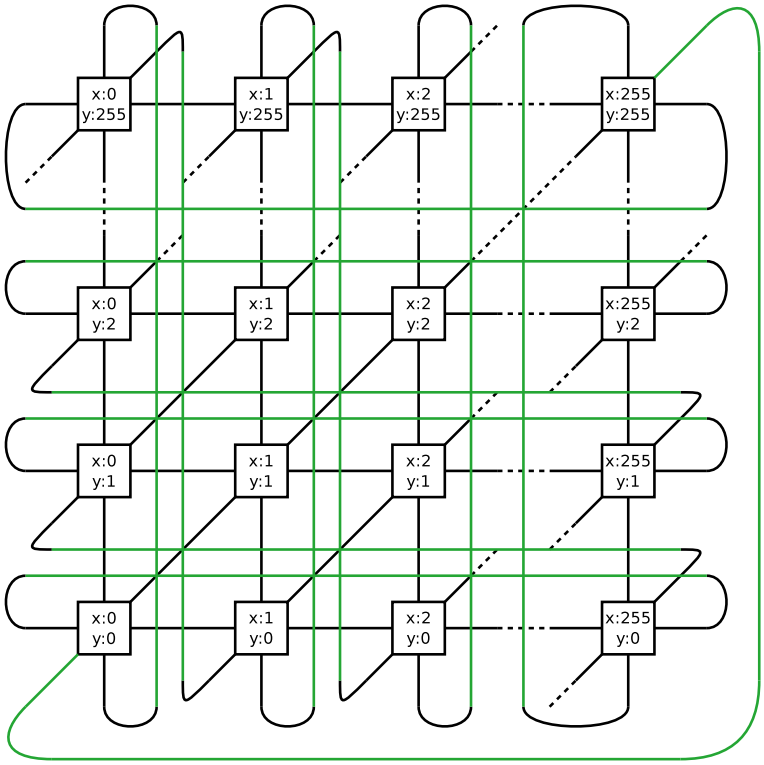


Figure 5.3: Two-dimensional grid of SpiNNaker chips with the needed connections (in green) to form the toroidal shape.

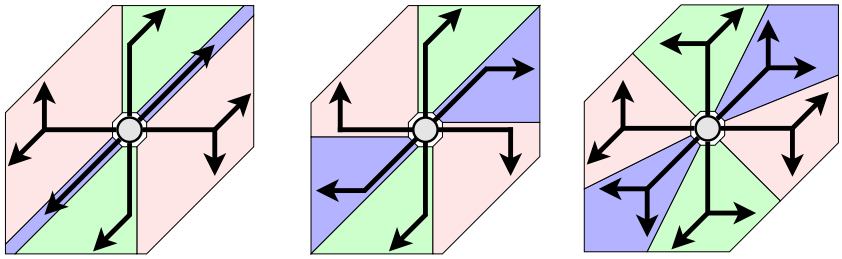


Figure 5.4: Shape of the routes generated with each algorithm.

5.5 Multicast route generation

Given that the generation of routes is outside of the scope of this chapter, only simple algorithms are considered for generating the multicast trees. All algorithms discussed in this section are well-known in the literature (Dally and Towles, 2004) and generate routes obliviously, *i.e.* they only require local information. This is a prerequisite for any multicast generation algorithm which has to be used in SpiNNaker because, for large configurations of the machine, each chip will be in charge of generating the multicast trees for its hosted neurons independently of the rest of the system. Furthermore, only the shortest path from a source to each destination is considered which, given the triangular topology (see Fig.5.3), will require, at most, advancing in two of the three possible directions (X, Y, Diag). The following algorithms will be taken into consideration in this study:

- Dimension Order Routing (DOR): This routes the packets first advancing all the required hops in the X dimension, then in the Y dimension and finally along the diagonal.
- Right Turn Only (RTO): Packets are routed in a way such that only right-turns are allowed. This require prioritizing the use of directions clockwise cyclically.
- Longest Path First (LPF): Each packet advances first through the direction having the largest number of hops and then in the other one (if any).

Fig.5.4 shows the shape of the routes generated by each algorithm and the areas covered by each local output port. In the figure, it is apparent that LPF has a better partitioning of the network and a more balanced use of network resources. To corroborate this, a simple experiment has been performed where $1,000$ random collections of

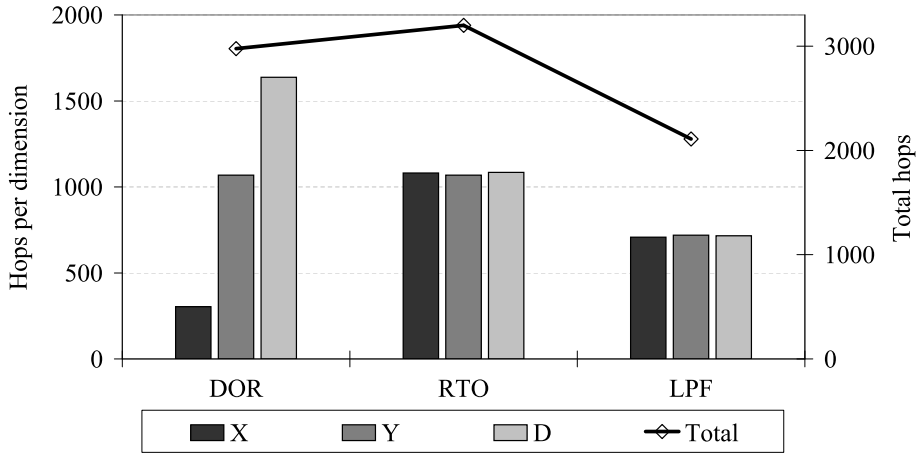


Figure 5.5: Network resources needed by each algorithm.

256 destination nodes were created and located, on average, 32 hops from the source. Then, using different algorithms, multicast trees have been generated and the network resources used by each algorithm measured. The results of this experiment are plotted in Fig.5.5 which shows the average resources used by each algorithm as well as the resource used per dimension; the latter provides an insight into the balance of the use of network resources. Note that an unbalanced use of network resources may lead to sub-optimal behaviour. The difference in terms of entries in the routing tables used by the different algorithms is negligible (less than 0.1%) and therefore will not be considered for deciding among them.

The results show that DOR has a very unbalanced use of network resources; it requires a greater use of the diagonal links (roughly 1.5 times Y and 5 times X) which are therefore likely to become a bottleneck. RTO provides a much more balanced use of the network, but at the cost of increasing the total network resources employed. However, as its use per dimension is noticeably lower than the most used dimension in DOR, it does compensate for the higher use of resources in other dimensions. Finally, LPF has the lowest requirements in terms of network resources (roughly $2/3$ of the other algorithms) and also produces a very balanced multicast tree. For this reason, the LPF algorithm has been implemented in SpiNNaker as a first approach to generate multicast routes and this algorithm is analysed throughout the rest of this chapter.

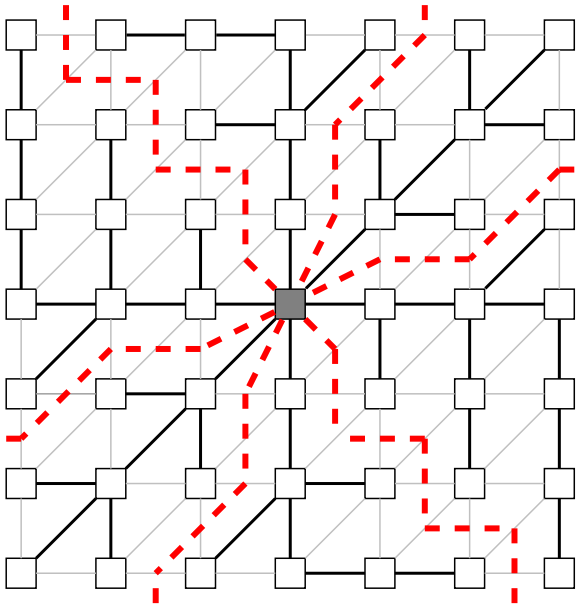


Figure 5.6: Links selectable by the "Longest Path First" algorithm.

5.5.1 Implementing LPF for the SpiNNaker system

The first step to compute the routing path is to evaluate the difference between the coordinates of the source and the destination chip. Doing so, the number of hops in the two main directions is determined: horizontal and vertical.

The second step is to compute the number of hops in the diagonal direction. The number of hops in this direction will be different from 0 only if the number of hops in the X and Y directions have the same sign (both positive or both negative). If diagonal hops are introduced, then the number of hops in the X and Y directions are re-computed accordingly. An example of the routing trees is described in Fig.5.6, the figure shows a 7×7 subnetwork with the beginning of the branches generated by this algorithm. Squares represent nodes, and lines between them represent the links; links which may be used by the multicast algorithm are darkened. The domains reachable from each output port are delimited by the red dashed line.

The routing paths for each of the source populations are computed step by step: each routing key is stored in a memory structure which reproduces the 256×256 structure of the SpiNNaker system, the routing algorithm then moves the packet forward by one step, storing the routing key in the appropriate memory structure. Finally all these (raw) entries are further processed so that if multiple entries belonging to the same routing keys are present in the memory structure representing one router, they

are compressed into a single entry with multiple destinations. At the same time, the default routing path is taken into account so that if an entry has to be default-routed, the entry will not appear in the final list of routing entries.

In addition, to avoid excessive memory consumption in the host during the computation of the routes, a "tiling" method has been applied: the whole 256×256 SpiNNaker chip assembly is divided into 64 blocks of 32×32 chips each of which is routed individually. The routing algorithm is applied to the connections which are sourced in each of the blocks of chips and the resulting entries are then compressed in the final list, freeing the memory for the next iteration.

5.6 Routing tests

The implementation of the novel population-to-population routing algorithm has been tested in three different scenarios:

1. One population per core, local projections only (recursive projections within each of the 16 cores of a chip used for neural computation);

2. One population per core, system-wide connectivity;

3. A simple biologically-inspired thalamocortical model.

Local projections are projections between populations of neurons allocated to the same chip. Long-range projections are projections between populations of neurons which do not reside in the same chip.

In the first two tests, the generated neural network consists of one population of 512 neurons for each of the cores in the complete SpiNNaker system: 16 cores are used for neural simulation in each of the 256×256 SpiNNaker chips of the whole system, for a total of $1,048,576$ populations and $536,870,912$ neurons.

Results have been obtained running the routing algorithm on an Intel Core 2 Duo T9600 (2.8 GHz) with 4 Gb of RAM using a Python program.

To appreciate the results presented in the following sections, it is important to compare them with the results obtained using a neuron-to-neuron routing approach. An example of such experiment has been published by Rast et al. (2011a), involving a network of $4,000$ neurons and $225,000$ synapses simulated on a 2×2 SpiNNaker chip test card (only one core per chip for neural simulation); this took about two hours (on a machine comparable to the one used for these experiments) to generate the complete

model which includes the neural model data structures, the routing table and the synaptic interconnection description. Since precise timing to generate such data structures is not available, here we estimate (optimistically) that the time required to generate such routing tables is only 10% of the full time ($\approx 12min.$). Based on this, a linear estimation (optimistic) of such a model, replicated on a 256×256 SpiNNaker chip system, each with 16 cores for neural simulation, is attempted. Such model would take

$$\frac{256 \times 256}{2 \times 2} \times 16 = 262,144$$

$262,144$ times longer than the one proposed in Rast et al. (2011a) to generate only the routing tables, $\approx 3,145,728$ minutes, which corresponds to ≈ 6 years.

5.6.1 One population per core, local projections only

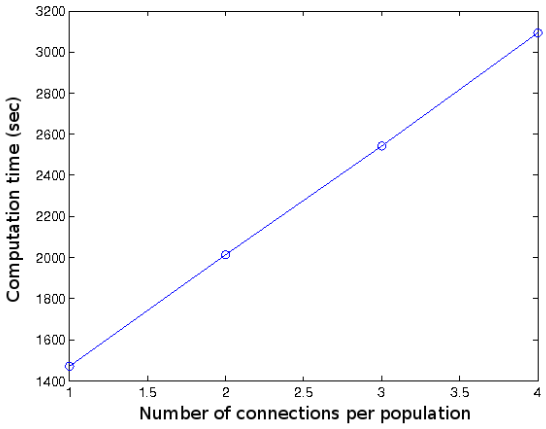


Figure 5.7: Results of the routing test with local projections only. The graph presents the linear relation between the number of connections per source population (inter-population connections) and the time required for the computation (in seconds).

The first test involved the generation of routing tables for a neural network where only projections within the same chip were present. These projections involve, therefore, a single hop through the local chip router to reach the destination core. In Fig.5.7 the relation between the time required for computation and the number of inter-population connections is presented. On the horizontal axis is the number of connections per source population (inter-population connections); on the vertical axis is the time required for the computation (in seconds). It is important to note that the computation time required is proportional to the number of connections defined.

5.6.2 One population per core, system-wide connectivity

Our second test involved the generation of routes for a complete SpiNNaker system. Every population selects, randomly, a collection of p destination populations distributed all over the system; note that self-connections and local connections are possible. In this case the time required for the computation and the number of routing entries generated represent the desired outcome of the test, therefore two graphs are presented: Fig.5.8 shows the computational time required to route the corresponding number of projections, Fig.5.9 shows the maximum and the minimum number of entries occupied in the routing tables.

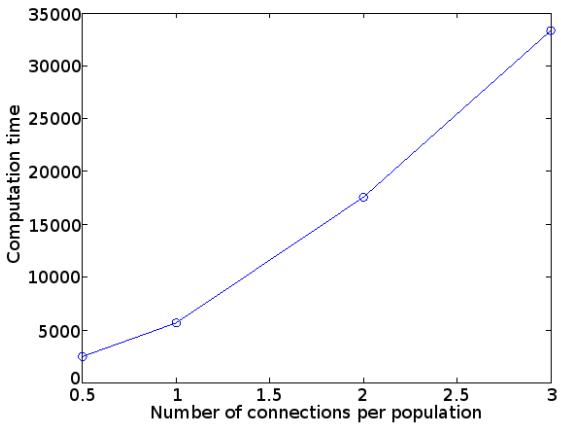


Figure 5.8: Results of the routing test with long-range projections only. The graph presents the relationship between the number of connections per source population (inter-population connections) and the time required for the computation (in seconds).

In both graphs, the horizontal axis represents the number of connections per source population (p). In the case of 0.5 connections per source population, only half of the populations in a chip have a projection towards another population in the system.

In Fig.5.8 the computational time is presented and from the graph it is clear the the computational time is no longer proportional the number of connections. Since single-hop routes require linear time, as shown by the previous test, the additional time is required to route the packets over multiple hops. Therefore the time required for the computation depends both on the number of connections sourced by a population, and on the distance (or equivalently on the number of hops) between the source population and the destination population(s).

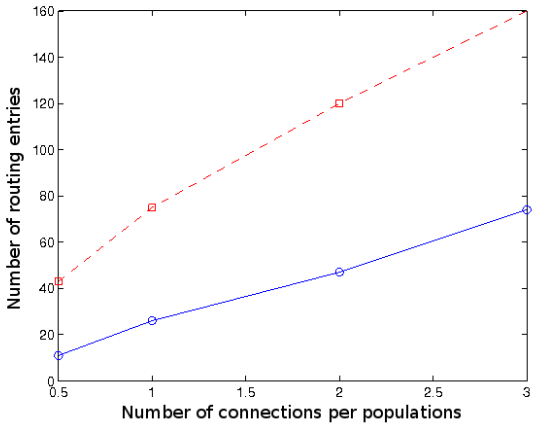
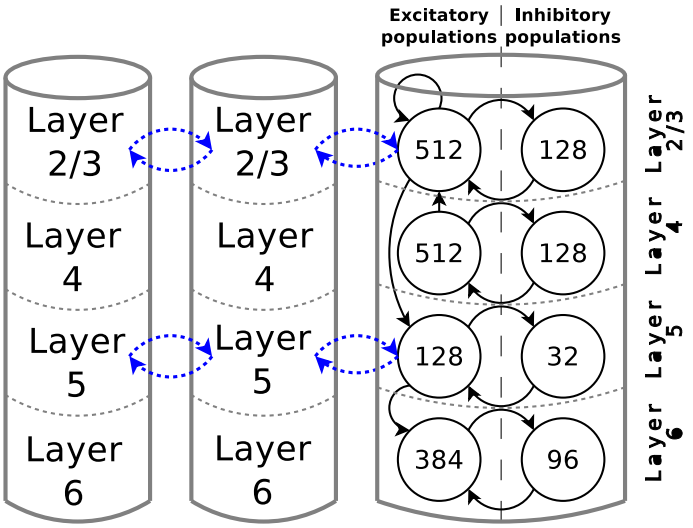


Figure 5.9: Results of the routing test with long-range projections only. The graph presents the relationship between the number of connections per source population (inter-population connections) and the number of entries in the routing tables. The blue solid line with circles represents the minimum number of routing entries. The red dashed line with squares represents the maximum number of routing entries.

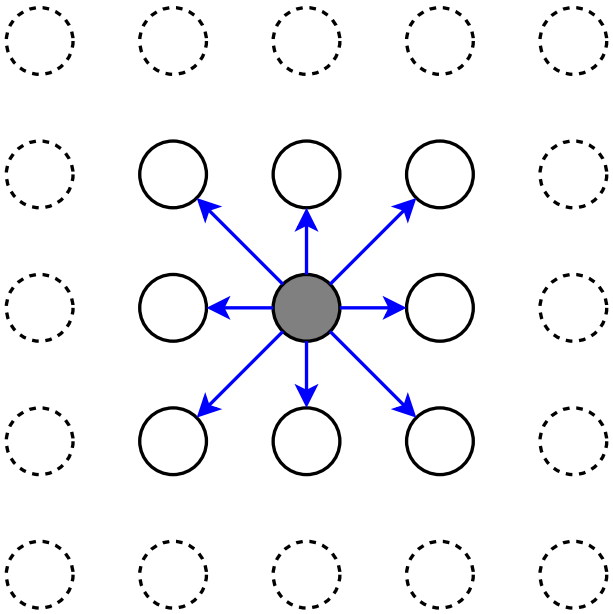
5.6.3 The thalamocortical model

In this last test a simplified biologically-inspired (Thomson and Lamy, 2007) thalamocortical model is used, as described in Fig.5.10(a). The figure shows the five-layer cortical columns considered in this study. A circle represents a population, and the number of neurons in the population is denoted within it. Black arrows represent short-range connections, those within a single cortical column; dotted arrows represent long-range connections between different cortical columns. A depiction of the structure of the long-range projections can be seen in Fig.5.10(b) in which circles represent cortical columns, and arrows represent projections among them. All the cortical columns follow the same structure of projections.

The system comprises 512×512 columns placed on a two-dimensional grid giving a total of $2,097,152$ populations of neurons ($503,316,480$ neurons) and $7,327,752$ projections (long and short range). The size of the greatest population has been selected according to the number of neurons which each core is able to simulate, given the interconnection pattern designed. However, since this model has not been simulated, this size is a speculation. Two factors make the choice of the population size unimportant from the point of view of this test. Firstly, the routing algorithm is independent of the exact size of the population, since the projection is represented by a single routing entry per population per projection. Secondly, cortical columns have



(a) A collection of cortical columns. The rightmost shows the populations within each column.



(b) Depiction of the inter-cortical-column projections.

Figure 5.10: Details of the thalamocortical model test.

been chosen because they offer a good demonstration for scalable systems, and provide a good model to exercise the capabilities of the routing algorithm. The model was planned so that four cores are required to simulate a single column:

1. Excitatory population of layers 2/3 - Total 512 neurons;
2. Excitatory population of layer 4 - Total 512 neurons;
3. Excitatory populations of layer 5 (128 neurons) and layer 6 (384 neurons) - Total 512 neurons;
4. Inhibitory populations of layers 2/3 (128 neurons), layer 4 (128 neurons), layer 5 (32 neurons) and layer 6 (96 neurons) - Total 384 neurons;

While short range connections are kept local to each chip (each chip is able to accommodate 4 columns which are allocated sequentially from the pool of columns to place), long range connections do not take full advantage of the regularity of the SpiNNaker architecture because the 512×512 columns grid does not match the 256×256 structure of the SpiNNaker system. However some sort of repeating pattern will be present for long range connections.

The routing algorithm generated the routing tables for the $65,536$ routers in the system using between 92 and 44 entries for each router. The histogram of the number of entries per router is described in Fig.5.11 where the horizontal axis represents the number of entries in the routers. The vertical axis represents the number of occurrences of the specific number of routing entries in the whole SpiNNaker system.

The routing tables for this experiment have been generated in slightly more than 7 hours using population-based routing principles.

5.7 Discussion

The three experiments described here provided the basis for an evaluation of population-based routing, with the particular goal of demonstrating that the time required to route all the projection. The number of entries required in each routing table, allow the problem to be solved in a time comparable to that required earlier using the neuron-to-neuron routing strategy, but on a much bigger system scale: 256×256 SpiNNaker chips are used during the experiments on the population-based routing, compared with 2×2 SpiNNaker chips used during the neuron-to-neuron experiments.

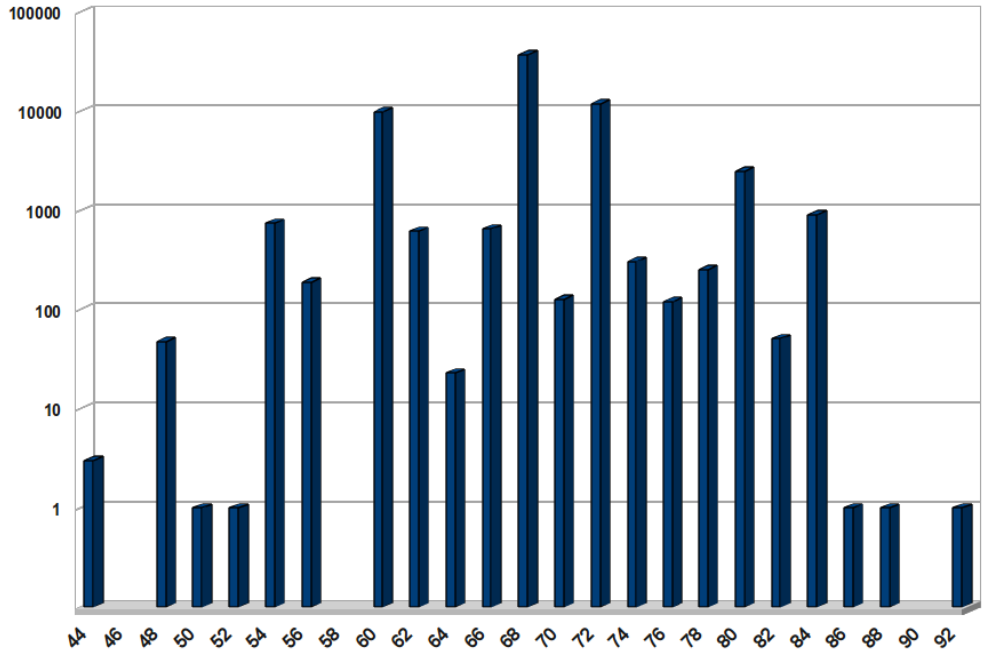


Figure 5.11: Histogram of the number of entries in the routing tables. On the horizontal axis the number of entries in the routing tables. On the vertical axis the number of occurrences for the particular number of entries.

In fact, previous results of routing neuron-to-neuron in SpiNNaker (e.g. Rast et al., 2011a), were obtained from a small neural network model ($4,000$ neurons and $225,000$ synapses, as described earlier), and the configuration of a system such as the biologically-inspired thalamocortical model described earlier would require ≈ 6 years to complete. With the proposed population-to-population routing approach the time required to generate the routing tables has been reduced to ≈ 7 hours, a reduction of several orders of magnitude.

The thalamocortical model experiment described earlier shows the result of routing using the principle that populations close together in the model reside in cores that are close to each other. Using a similar approach, also, for long-range connections may further reduce the number of entries in each router. In addition, in the case that the number of entries required is greater than the size of the routing table, it is possible to compress the entries further using a logic minimization tool (e.g. "espresso" or "ROCM", Lysecky and Vahid, 2003). This type of software automatically minimizes boolean logic functions so as to use a smaller number of gates. The multicast routing table represents a (complex) form of boolean functions, for which it is also possible to specify the "don't care" bit(s) for each entry using the router's memory appropriately

(see section 3.3.3).

A minor drawback with this algorithm is a need for an addressing space which is larger compared with the standard neuron-to-neuron communication channel definition (in the worst case the addressing space required is double the space strictly needed for the number of neurons simulated). However the SpiNNaker system has been designed to simulate a 1 billion neuron neural network, which is only about 25% of the addressing space offered by a 32 bit system, giving some additional space for a rational organization of the routing keys.

This chapter has described the multicast packet router in the SpiNNaker system and how the routing tables can be computed off-line. This process, in practice, is now hosted on a PC which computes the routing tables starting from a high-level description of the neural network. In this context the "Longest Path First" algorithm has been chosen because, between the algorithms which require only local information to route packets to the destination chips (this algorithm requires only the source chip coordinates and the destination chip coordinates), it is the one which uses the minimum network resources and has the most balanced usage of network links. The routing path may be computed hop by hop by each of the routers through which the packet has to pass, until the destination router is reached.

The outcome of this routing algorithm, however, shows that for both random networks and biologically-inspired networks, the number of entries used in the routing tables is rather low, compared to the number of entries available. Therefore, even if the models used are designed only for testing purposes, the routing entries which are still available for each router enable users to design networks which are more biologically detailed.

The new approach moves the problem of the distribution of the spikes in the system up of one level of abstraction: the number of projections does not depend any more on the size of each single population, but on the number of the populations simulated in the system. The computational improvements in this approach have allowed the routing of projections across a very large neural network, such as those for which the SpiNNaker system has been designed, using a standard PC in a reasonable time.

The content of the last chapter (the SpikeServer implementation) and this chapter are applied together in the next chapter, which contains the main contribution of this work: the STDP-TTS learning rule.

The algorithm described here is the first step towards a greater goal: a self reconfiguring neuromorphic architecture; it has been designed to allow an implementation

directly on the SpiNNaker system. In this way, during the setup phase, the SpiNNaker system will self-configure determined by a high-level description of the neural network. The routing algorithm described is only a first step, further steps include the generation of each synaptic connection required by the high-level description, so that the SpiNNaker system has the features of a self-configuring architecture.

Population-based routing moves the SpiNNaker architecture forward one step towards synaptic rewiring (also known as structural plasticity): during the neural simulation, when required from the biological model, it will be possible to connect two populations which initially were not connected. This is a biological process which happens naturally at the beginning of life and with the acquisition of experience, but that has not yet been completely understood.

The SpiNNaker system has been designed to provide a tool for biologists to perform experiments and test hypothesis which emerge from the study of biological neural networks. The tool presented here is a step towards the implementation of such models in the SpiNNaker system with the objective of providing a tool to simulate such biological processes, abstracting the description of the neural network up by one level to the population scale.

5.8 Summary

This chapter presented the second of the author's contributions: the population-based routing approach. The idea presented here, together with the implementation of the interface to inject input spikes into the SpiNNaker system (the SpikeServer) have been used in the process of testing the novel learning rule, the STDP-TTS, which will be presented in the next chapter, and describes author's main contribution.

Chapter 6

The STDP-TTS learning model

6.1 Introduction

This chapter introduces the main contribution of this research work: the STDP TTS learning rule. The presentation of this research topic provides an introduction and background section in which a brief overview and history of synaptic plasticity is presented. After a brief overview of the technical details of the SpiNNaker system used across this chapter, section 6.3 discusses the motivation for this research. Subsequently the extraction of some statistical features, used for this learning rule, is described. Finally the characterization of the proposed learning rule goes through various steps in which the results are compared with the standard STDP and the spike-pair STDP learning rules implementation on SpiNNaker. Results are presented to show how each parameter of the learning rule influences the behaviour of the STDP TTS algorithm. The task used to characterise this learning rule is a pattern recognition task: a repeating pattern (randomly generated) hidden in a background noise is presented as input to a neural network. As a consequence, the output neuron should emit an action potential every time it identifies this pattern. Multiple patterns interleaved are also used to check the capacity of the network (and of the learning rule) to adapt and to identify multiple patterns.

6.2 Background

6.2.1 Introduction to synaptic plasticity

Artificial neural networks increasingly involve spiking dynamics to permit greater computational efficiency. This becomes especially attractive for on-chip implementation using dedicated neuromorphic hardware. However, both spiking neural networks and neuromorphic hardware have historically found difficulties in implementing efficient, effective learning rules. The best-known spiking neural network learning paradigm is Spike Timing Dependent Plasticity (STDP) (Markram et al., 2012) which adjusts the strength of a connection in response to the time difference between the pre- and post-synaptic spikes. Approaches which relate learning features to the membrane potential of the post-synaptic neuron have emerged (e.g. Clopath et al., 2010) as possible alternatives to the more common STDP rule, with various implementations and approximations. Here a new type of neuromorphic hardware is used, SpiNNaker, which represents the flexible "neuromimetic" architecture, to demonstrate a new approach to this problem. Based on the standard STDP algorithm with modifications and approximations, a new rule, called STDP TTS (Time-To-Spike), relates the membrane potential with the Long Term Potentiation (LTP) part of the basic STDP rule. Meanwhile, the spike-pair STDP rule is used for the Long Term Depression (LTD) part of the algorithm. On the basis of the membrane potential it can be shown that it is possible to make a statistical prediction of the time needed by the neuron to reach the firing threshold, and therefore the LTP part of the STDP algorithm can be triggered when the neuron receives a spike. In our system these approximations allow efficient memory access, reducing the overall computational time and the memory bandwidth required. The improvements here presented are significant for real-time applications such as the ones for which the SpiNNaker system has been designed. Simulation results show the efficacy of this algorithm using one or more input patterns repeated over the whole time of the simulation. On-chip results show that the STDP TTS algorithm allows the neural network to adapt and detect the incoming pattern while reducing the amount of computation required. Through the approximations suggested in this chapter, a learning rule is introduced which is easy to implement both in event-driven simulators and in dedicated hardware, reducing computational complexity relative to the standard STDP rule. Such a rule offers a promising solution, complementary to standard STDP evaluation algorithms, for real-time learning using spiking neural networks in time-critical applications.

6.2.2 Learning in biological neural networks

Biological neural networks are known for their ability to learn and neuroscience research has shown that the learning abilities of a neural network are connected with the plasticity of the neural network itself (Cajal, 1894). Hebb (1949) proposed the first description of a mechanism behind the innate capability to learn, frequently summarized as: "Cells that fire together, wire together". This general rule emphasises the feature of causality in the activation of neurons: if the pre-synaptic neuron fires an action potential, and as a consequence of this event, the post-synaptic neuron also produces an action potential, the connection (synapse) between the two neurons should be potentiated. However, Hebb's basic postulate takes into account only the *increase* in synaptic efficacy, disregarding the possibility of a *decrease* in synaptic efficacy. Furthermore, this possibility has been analysed in theoretical work and found to be important to avoid saturation of the synaptic efficacy (Sejnowski, 1977; Bienenstock et al., 1982). Likewise it is critical if the potential fidelity of the "recall" in associative memory is to be maximized (Willshaw and Dayan, 1990).

Two different phenomenological descriptions of an undelying mechanism (processes) can account for long-term synaptic depression: the first and most well-known is related to the relative timing of the pre- and post-synaptic spikes (STDP). The second process is called "homeostasis" and happens following a synaptic strengthening event: after synaptic potentiation, all plastic synapses afferent on a neuron should be re-normalized so that the total amount of input to a neuron remains constant (Sejnowski, 1977; Malsburg, 1973). Such re-normalisation is easy to achieve in a model, where each node has global visibility of its input weights, but more complex to justify in biology where, since each synapse only has local visibility, there must either be some neuromodulatory intermediary to act as a local proxy, or a specific structure in the dynamics which conveys global information through local input spikes. Although these processes imply that the information is stored in the timing of the spike event, Roberts and Bell (2002) highlight a point of contention in neuroscience: it is not yet clear whether information is encoded in the precise timing of the events or in their rate.

The underlying biological details of STDP have been well-studied. In 1996 Markram and Tsodyks (1996) showed that synaptic weight modification occurs when pre- and post-synaptic spikes coincide at low frequencies. More results were described by Bi and Poo (1998) after tests run on hippocampal cells: the results published show a relation between synaptic weight modification and the relative timings of pre- and post-synaptic spikes. The curve in Fig.6.1 shows a functional approximation to the

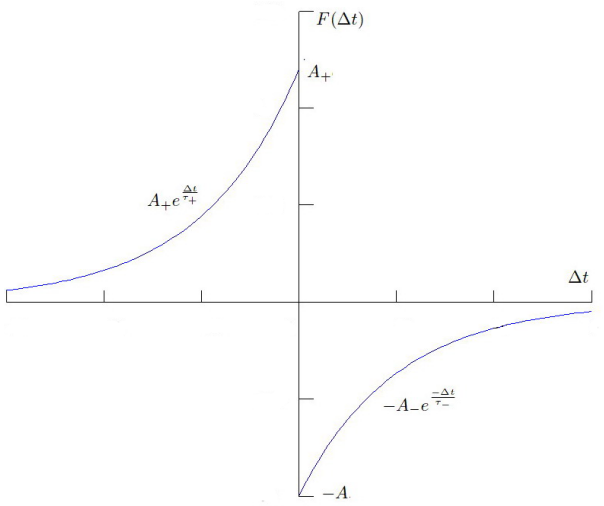


Figure 6.1: STDP curve. The horizontal axis represents the time between pre- and post-synaptic spikes ($\Delta t = t_{pre} - t_{post}$). The vertical axis represents the synaptic weight modification.

fitted correlation, which is called the STDP curve, and is represented by the following two equations:

$$\Delta w = F(\Delta t) = \begin{cases} A_+ e^{\frac{-\Delta t}{\tau_+}} & \Delta t \geq 0 \\ -A_- e^{\frac{\Delta t}{\tau_-}} & \Delta t < 0 \end{cases} \tag{6.1}$$

where $\Delta t = t_{post} - t_{pre}$, and A_+, A_-, τ_+, τ_- are parameters of the learning function. The output of the function is the weight variation.

These experiments, however, relate only *pairs* of pre- and post-synaptic spikes; it is not yet clear how multiple sets of pre- and post-synaptic spikes influence synaptic weights. Furthermore, the efficacy of the synaptic weight modification has since been shown to be limited by an efficacy time window, and the precise values of the curve vary depending on the system and the neuron type (Abbott and Nelson, 2000; Bi and Poo, 2001; Song and Abbott, 2001).

Graupner and Brunel (2012) propose a calcium-based model that, in their view, justifies the known biological results extracted through experimentation. This model consists of a single update rule of the synaptic efficacy which is able to reproduce the results obtained from the original experiments by Bi and Poo (1998) which led to the original description of the standard STDP curve (see Fig.6.1), as well as the results obtained from the experiments which led to the triplet- (Pfister and Gerstner, 2006)

and quadruplet-based (Wang et al., 2005) update rules. In addition the newly proposed rule takes into consideration also the case of spike bursts which determine variations in the synaptic efficacy.

Biological neural networks involve an important trade-off between two competing processes (Turrigiano, 1999): the first is the need for change, the second the need for stability. Carpenter and Grossberg (1987) identify this clearly in an important early example of (non-spiking) winner-take-all networks as the "stability-plasticity trade-off". The first process allows changes which modify the activity pattern in neural networks, creating progressively more emphasized differences between neurons, but with the effect of destabilizing it (Kube et al., 2008). The second process stabilizes the activity in the neural network, making the activity uniform. In particular, the first process is associated with learning novel inputs, while the second normalizes the input of a neuron on the basis of its mean firing rate, a state termed "homeostatic equilibrium". Gilson et al. (2009) make an extensive analysis of this trade-off for a general neural network and find that the network structure is critical: in order to obtain stable networks which can successfully learn differentiated patterns, the inputs must be arranged in correlated input groupings, while the initial distribution of weights on the input neurons must be non-uniform. Grossberg and Versace (2008) take a more explicit approach, significantly extending and transforming the ART (Adaptive Resonance Theory) model into a spiking representation incorporating the same tradeoffs. As per earlier ART models this uses an auxiliary resetting network to detect novel inputs and make category groups eligible for learning. Both of these approaches emphasize the need for non-uniform initial network structure.

At a network level, Hebb's general postulate has led to a variety of "Hebbian" learning rules. While initially the primary attraction of Hebbian rules was their ability to support unsupervised learning, the discovery of STDP in biology - and its subsequent identification as a Hebbian process (Gerstner and Kistler, 2002) - provide new motivation by suggesting or explaining biological learning mechanisms. Various different learning rules have been examined theoretically and in simulation in an attempt to find biologically realistic models. The field of neural coding provides several important examples of the use of STDP in practice. The timing sensitivity of STDP provides an obvious mechanism for network synchronisation. Izhikevich (2006) has proposed several synchronous networks based upon STDP which use it to tune neurons with convergent delays to form timing-locked logical groups. However, while synchronous

networks offer a *dynamic* explanation of biological patterns, they have not thus far provided a complete *functional* explanation - that is, the role of synchronisation in the processing remains disputed (Masuda and Kori, 2007). Winner-take-all (WTA) networks provide explicit functional coding - either for the formation of stimulus-selective cortical columns (Song and Abbott, 2001), or for methods of pattern selection (Masquelier et al., 2008a). Rank Order Coding (Thorpe and Gautrais, 1998) may be considered as a form of time dependent WTA neural coding, where information is encoded in the order in which a neuron's inputs fire. STDP has been used in conjunction of Rank Order Coding in the emergence of selective orientation receptive fields (Delorme, 2001) and in the unsupervised learning of visual features (Masquelier and Thorpe, 2007). Reinforcement learning models prescribe even more explicit functional roles, such as the use of STDP to construct a predictor (Rao and Sejnowski, 2001). Critically, STDP remains, to date, the only biologically plausible underlying mechanism to demonstrate functional learning rules for spiking neural networks (Arena et al., 2007) and, as such, forms a central part of spiking models.

6.2.3 Algorithmic techniques for synaptic plasticity

When modelling neural networks in simulation, it is necessary to determine a precise form for the learning rule. Multiple learning algorithms have emerged, attempting various trade-offs between computational complexity, biological realism, and analytical tractability. The best-known Spike Timing Dependent Plasticity rule considers every possible combination of pre- and post-synaptic spikes, modifying correspondingly the synaptic weight (Song and Abbott, 2001) (see Algorithm 6 for pseudo code of this learnign rule). The STDP has been used in several tests (e.g. Guyonneau et al. (2005)) and has proven to replicate the biology with some degree of accuracy. Burkitt et al. (2007) have given a mathematical and statistical characterization of this rule. One particular property of the STDP emerges from the tests of Guyonneau et al. (2005): a neuron which identifies an input pattern repeatedly advances the timing of identification of this pattern earlier and earlier, until it reaches the earliest group of spikes that begins the pattern.

An approximation of this algorithm is spike-pair (or nearest-neighbour) STDP. This is the simplest learning rule which has shown some biological realism in experiments (Masquelier et al., 2008b,a). It is known to approximate at least one biologically relevant feature suggested by Izhikevich and Desai (2003): post-synaptic spikes propagate back into the dendritic spines, resetting them. As a result, the latest post-synaptic spike

Algorithm Name: SPIKE TIMING DEPENDENT PLASTICITY
Description: This routine described the standard STDP rule, which modifies the synaptic weights pairing the time of all the pre-synaptic spikes with the time of all the post-synaptic spikes.

---

foreach *pre-synaptic spike* **do**
  foreach *post-synaptic spike* **do**
    $\Delta t = Time_{pre} - Time_{post}$;
    Δw =*compute_weight_modification(*Δt*)*;
    new_weight = *update_synaptic_weight(current_weight,* Δw*)*;
  end
end

Function description:
$Time_{pre}$: Time of the pre-synaptic spike;
$Time_{post}$: Time of the post-synaptic spike;
compute_weight_modification(): computes the weight modification according to the STDP curve (Fig.6.1);
update_synaptic_weight(): updates the synaptic weight according to the current weight and the weight modification previously computed, applying also hard limits on the synaptic weight value, where appropriate;

Algorithm 6: The Spike Timing Dependent Plasticity (STDP) algorithm.

"erases" the effect of earlier pre-synaptic spikes. This rule is frequently implemented in analogue hardware neural network chips (e.g. Indiveri et al. (2006)) since it is simple: the timing of only one pre- and post-synaptic spike is needed to trigger synaptic potentiation and depression, eliminating the need for complex, area-intensive spike buffering.

Bienenstock et al. (1982) have created a theoretical learning rule (called the BCM rule after the authors' initials): the basis of this algorithm is that the instantaneous firing *rate* rather than individual spikes (as for STDP) set the pattern of weight modifications. Synaptic inputs which drive the post-synaptic neuron to high firing rates are potentiated, while inputs which produce low firing rates are depressed. Izhikevich and Desai (2003) have proven an equivalence of the BCM rule and the STDP spike-pair rule, opening both to common theoretical treatment and adding flexibility in specific model implementation choices.

More variants of the STDP rule have been presented which take into account trains of three and four pre- and post-synaptic spikes (Pfister and Gerstner, 2006; Wang et al.,

2005). These rules try to explain a biological effect which shows synaptic depression when neurons are subject to high input firing rates, where standard STDP would potentiate those synapses (Sjöström and Gerstner, 2010). In particular, in Pfister and Gerstner (2006), a novel description of synaptic plasticity based on triplets of spikes seems to reproduce neurobiological experimental results with more accuracy, particularly under conditions where the standard STDP rule shows limitations. The described behaviour seems to match the BCM rule described before (Pfister and Gerstner, 2006), even if the mathematical proof is limited by some statistical constraints on the input.

The algorithms which directly relate spike timings to synaptic weight modification have been shown, more or less, to be biologically plausible. Recently, a new type of algorithm based on the use of other observables or model variables as proxies for the spike timing has started to emerge. Such algorithms dramatically reduce the computational load since the model does not have to compute each independent spike time. One new class of those, in particular, relate the synaptic weight modification to the (pre- or post-synaptic) neuron membrane potential (or by a low-pass filtered version of it) (e.g. Clopath et al., 2010; MacNeil and Eliasmith, 2011). Artola et al. (1990) describe the biological principles supporting this class of algorithm: depending on whether the membrane potential of a post-synaptic neuron is above or below an LTP or LTD threshold, the appropriate update is triggered. This assumes a form of voltage gating of the synaptic update, much like the voltage gating term for transmission of NMDA synapses.

In this context a new algorithm has been proposed by Davies et al. (2011a) which relates the synaptic weight modification with the post-synaptic membrane potential in a network of Izhikevich neurons. The paper presents an implementation of the STDP TTS (time-to-spike) rule in a Matlab environment, and it shows that the new algorithm is able to detect one or multiple patterns in a noisy background, lock the identification and tune to the earliest possible group of spikes. This algorithm has been reproduced on the SpiNNaker hardware (Davies et al., 2012a) and presented similarities with the Matlab implementation, even though there were numerical differences between the two implementations. In this chapter an implementation of the STDP TTS learning rule on the SpiNNaker neuromimetic simulator is presented. It has improved since that presented in (Davies et al., 2012a), and it is compared it with the best-known standard STDP and the spike-pair rules to show that it is less expensive from both a computational and a memory utilisation aspects, while being able to detect patterns with performance comparable to the standard STDP rule.

6.3 Architecture of the SpiNNaker system

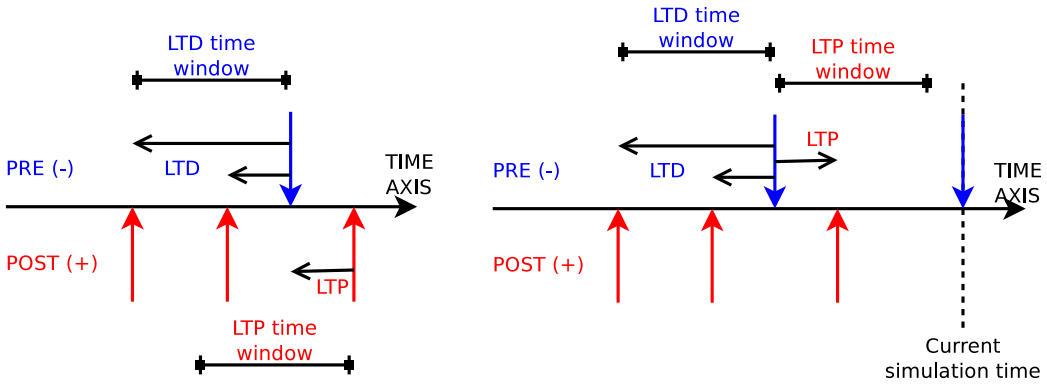
The hardware and software architectures of the SpiNNaker system have been described in chapter 3. This chapter focuses on the learning rules available, and provides an extension to the ones already implemented.

Learning rule

The current learning rules developed for the SpiNNaker simulator are based on the standard STDP and its spike-pair variant algorithms. Usually, in software simulators, the standard STDP rule is implemented so that, when a spike is received by the post-synaptic neuron, LTD is triggered with respect to all the spikes that the destination neuron has emitted in the relevant time window. Equivalently, when an action potential is emitted by a neuron, LTP is triggered with respect to all the incoming spikes from all the synapses (see Fig.6.2(a)). A similar mechanism is commonly used for the spike-pair STDP rule.

In the SpiNNaker system such implementation is not easily programmable: the synaptic data structures are available to process only when a spike is received. Therefore a novel model has been implemented (Jin et al., 2009, 2010c) to store all information related to the incoming and outgoing spikes and trigger LTD and LTP when all the relevant information is available locally in the core (see Fig.6.2(b)). In this example, LTP and LTD are both triggered when an incoming spike is received: the STDP time window for the previously received spikes is closed, meaning that all the relevant post-synaptic spikes are available, and therefore it is possible to compute both LTP and LTD. This execution model has been called the Deferred-Event Model (Rast et al., 2009).

This type of implementation has an impact on the memory occupied by the data stored for the records (incoming spike record and outgoing spike record), and on the computation required to perform the synaptic weight update; these impacts are described in the cited articles. The research topic proposed in the remainder of the chapter originates from the large requirements in computational complexity and memory occupation for the implementation of the standard and the spike-pair STDP rules.



(a) Standard STDP software implementation

(b) STDP with Deferred Event Driven Model

Figure 6.2: Implementation of the standard STDP learning rule.

6.4 The STDP TTS learning rule

The key feature of this new learning algorithm is that it is able to update the synaptic weight as soon as a pre-synaptic action potential is received by the post-synaptic neuron. To achieve this, LTD is triggered using the spike-pair STDP rule, while for LTP the synaptic modification is triggered in relation to the membrane potential value of the post-synaptic neuron.

It may be relevant to note the difference from the standard STDP rule: the STDP TTS does not require an action potential emitted by the post-synaptic neuron to trigger LTP on the synapse from which an action potential is arriving.

To perform this algorithm, a relationship between the membrane potential and the estimated time to spike of a neuron is needed. To determine this, a statistical approach is used: the basis is that the higher the membrane potential, the sooner the neuron is going to fire (with some probability). This forecast model is called "Time-To-Spike" (TTS), and the LTP part is similar to the spike-pair STDP rule, although the time of the outgoing spike is unknown at the moment of the weight modification, and is estimated on a statistical basis.

In contrast to the standard STDP model implemented in the SpiNNaker system, which requires the Deferred Event Model as described above (Jin et al., 2010c, 2009), this algorithm computes both potentiation and depression (LTP and LTD) when a spike arrives at its destination.

6.5 Statistical feature extraction

For the purpose of this chapter we focus on Izhikevich neurons, described earlier in section 3.6.1. Using a network comprising $4,000$ Izhikevich neurons randomly interconnected with random delays the relation between the membrane potential and the time required for this neuron to fire is estimated. Two types of neurons are used in this network:

1. Tonic spiking neurons: $(a = 0.02; b = 0.2; c = -65; d = 8)$;
2. Fast spiking neurons: $(a = 0.1; b = 0.2; c = -65; d = 2)$;

There are $3,200$ neurons of the first class each connected to 25 neurons of both classes with excitatory synapses. The synaptic strength of these connections is $10mA$. In addition there are other 800 neurons of the second class each connected to 25 neurons randomly chosen between the tonic spiking neurons with inhibitory synapses. The strength of these connections is $-5mA$.

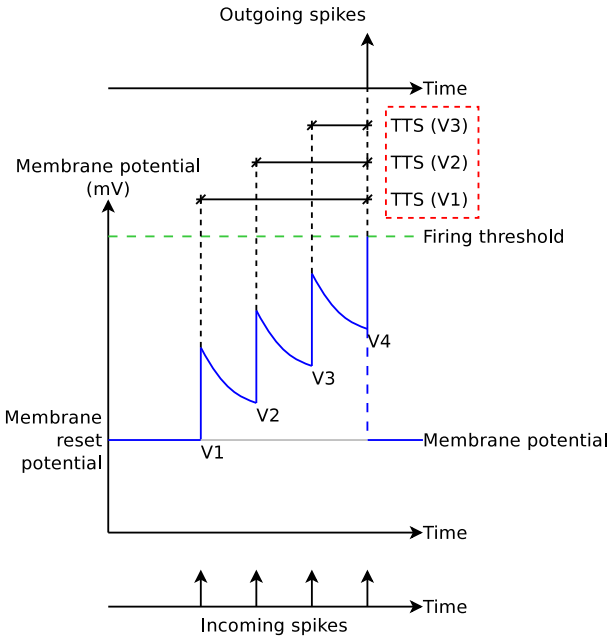


Figure 6.3: Example of computation of the Time-To-Spike (TTS) of a neuron.

The whole network is stimulated by the injection of a constant current of $20mA$ into 60 randomly chosen neurons in the first class and 20 randomly chosen neurons in

the second class. No synaptic plasticity is enabled for this test. The simulation is run for 1000 steps of $1msec$ each, for an overall simulation time of $1sec$.

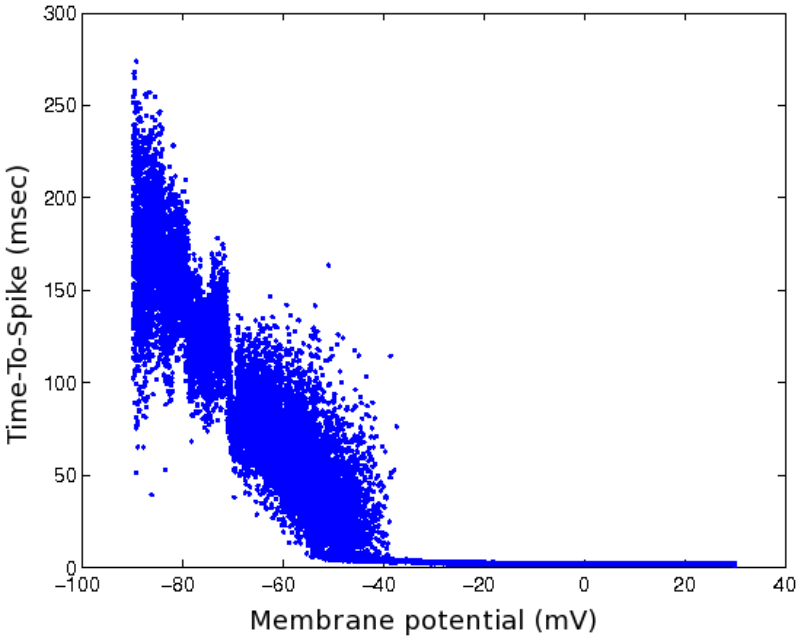
During the simulation, the software records the membrane potential for all the neurons in both populations and the incoming spike times. At the end of the simulation the membrane potential record is post-processed: for every spike received by a neuron a couple (membrane potential, time-to-spike) is computed (see Fig.6.3). Finally the computed couples are grouped and sorted on the basis of the membrane potential, and a mean value for the time-to-spike between all the elements in a group is computed, and it is presented in Fig.6.4(a).

The difference between the two parts of the graph in Fig.6.4(a) can be explained by examining the Izhikevich neuron phase space (Fig.6.5). If the value of the neuron state variables v and u are in the half-plane on the left of the separatrix (the dashed line and the BC part of the parabola in the graph), the neuron state will move towards equilibrium point A in the absence of input. By contrast, if the neuron state is in the right half-plane, the neuron state will move towards the firing condition ($v \geq 30mV$). The threshold that identifies the two half-planes corresponds to the discontinuity between the two sections of the graph in Fig.6.4(a).

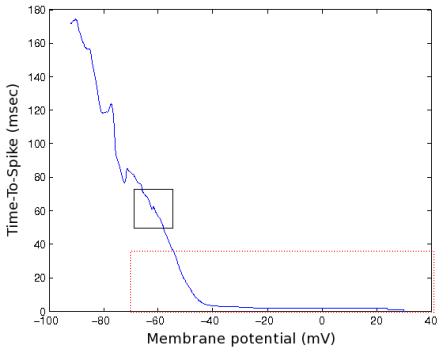
6.5.1 Post-processing of the statistical function

To remove some noise from the graph in Fig.6.4(a),a sliding window filter has been applied over an interval of $1,024$ equally-spaced membrane potential values, so that variation of the time-to-spike in the membrane potential/time-to-spike function is smoother. The outcome of this post-processing is shown in Fig.6.4(b). While the section on the right side of the graph is linear as described before, the section on the left shows some fluctuations. In particular, around the point $(-60mV; 60msec)$ (see detailed Fig.6.4(b)) it is possible to note the first evident non-monotonicity: in a short interval around those values, for lower values of the membrane potential the neuron is predicted to fire sooner. This is interpreted as a statistical aberration, and towards the left part of the graph these aberrations become more evident. To avoid encountering such regions a window of $32msec$ is selected (Fig.6.4(c)).

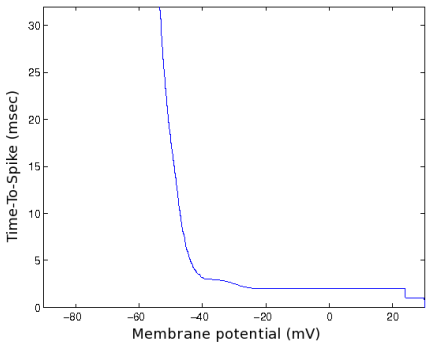
The figure shows a function characterized by two segments: the first passes through the points $(30mV; 0msec)$ and $(-40mV; 3msec)$; the second through the point $(-40mV; 3msec)$, and the second edge of the segment is used as a parameter for the forecast (parameter "L"). The point $(-40mV; 3msec)$ is actually a hinge point for the second segment, and it is kept constant as the separatrix in the Izhikevich state phase



(a) Raw version of the relation function. The graph can be divided into two sections with very different behaviours, the discontinuity occurring at about $-40mV$. For membrane potentials above this value, estimated time to spike is linear between $3msec$ and $0msec$. For values below $-40mV$ the graph is very noisy.



(b) Filtered version of the relation function. The filter applied is a sliding window over an interval of 1024 equally-spaced membrane potential values. The square highlights the first incongruence in the forecast of the Time-To-Spike: in this region the lower the membrane potential, the sooner the neuron fires. The red dotted line identifies the area considered for the forecast function.



(c) Enlargement of the $32msec$ time window used for the forecast function, highlighted with the red dotted line in the previous image.

Figure 6.4: Function that relates the membrane potential in mV (on the horizontal axis) and the estimated time to spike in $msec$ (on the vertical axis).

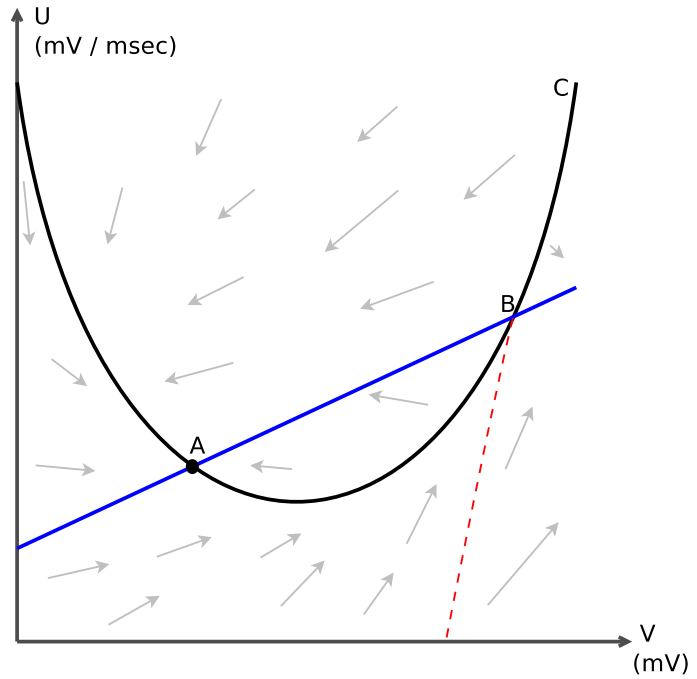


Figure 6.5: Izhikevich neuron state phase plane The horizontal axis is the membrane potential variable v. The vertical axis is the membrane recovery variable u. The black parabola represents the nullcline for the $\dot{v}$ equation. The blue line represents the nullcline for the $\dot{u}$ equation. The red dashed line represents the separatrix between the attraction domain of the equilibrium point A (left of separatrix) and the domain where the neuron reaches eventually the spiking condition (right of separatrix).

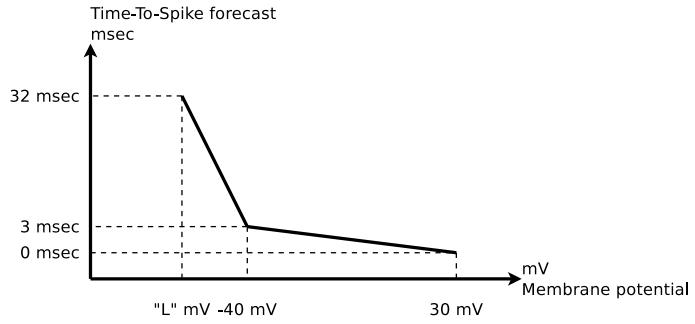


Figure 6.6: Relation between membrane potential and time to spike of the neuron.

plane is constant. Fig.6.6 summarises the relation function between the membrane potential and the estimated time to spike.

6.5.2 The STDP-TTS learning algorithm

The algorithm which is performing the proposed learning rule is summarized with the pseudo-code of Algorithm 7. As this algorithm is similar to the spike-pair STDP rule, the first part of the algorithm performs the Long-Term Depression (LTD) part of the algorithm using the time of the last known spike emitted by the post-synaptic neuron. The difference of these time stamps are used as input to the STDP function (Fig.6.1). The second part of the STDP TTS algorithm performs the forecast and the Long-Term Potentiation (LTP) part of the STDP algorithm, using again the STDP curve to evaluate the variation of the synaptic weight. In particular the forecast is computed on the basis of the post-synaptic neuron membrane potential, applying the function described earlier.

Figure 6.7 shows an example of the behaviour of this learning rule: only the synapse carrying the last spike is potentiated because the membrane potential at the onset of the spike is above the learning threshold.

6.6 Network description and input generation

To verify this learning rule, to test performance and to compare results with other learning rules, the network structures depicted in Fig.6.8(a) and Fig.6.8(b) are used.

The network in Fig.6.8(a) is used to test the behaviour of the learning rule in the case of a single input pattern immersed in the noise. The second network (Fig.6.8(b))

Algorithm Name: THE STDP-TTS LEARNING RULE
Description: This routine performs synaptic weight modification according to the Spike Timing Dependent Plasticity with Time-To-Spike forecast (STDP-TTS) rule.

---

compute_LTD(previous_post_synaptic_spike_time, current_time);
m_p = *retrieve_post_synaptic_neuron_membrane_potential()*;
t_t_s = *forecast_tts(m_p)*;
compute_LTP(current_time, t_t_s);

Function description:
compute_LTD(): computes Long-Term Depression using the previous known post-synaptic spike time and the current simulation time;
compute_LTP(): computes Long-Term Potentiation using the current simulation time and the forecasted value for the post-synaptic spike;
retrieve_post_synaptic_neuron_membrane_potential(): retrieves the post-synaptic neuron membrane potential;
forecast_tts(): estimates the time of the future post-synaptic spike based on the post-synaptic neuron membrane potential;

Algorithm 7: The STDP-TTS algorithm.

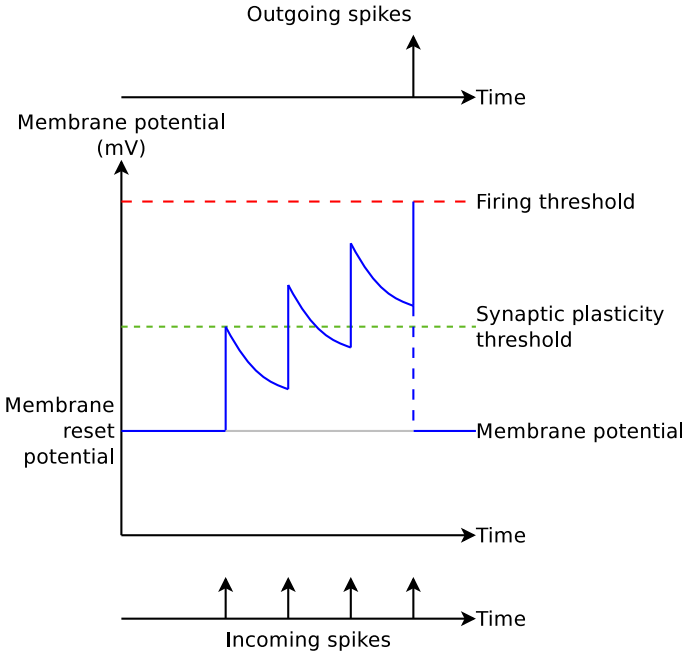
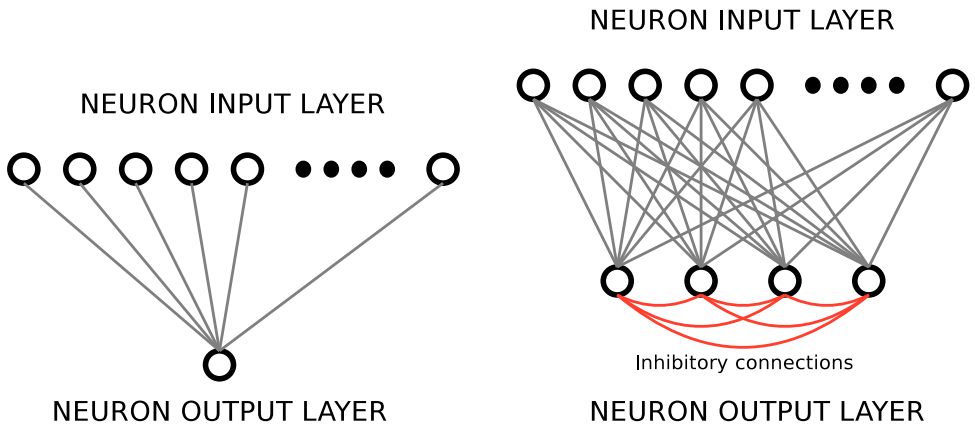


Figure 6.7: Depiction of LTP trigger mechanism.



(a) Network structure: 800 input neurons - 1 output neuron

(b) Network structure: 800 input neurons - 4 output neurons. The output neurons inhibits each other (red links).

Figure 6.8: Structure of the neural networks used in the tests: 800 input neurons and 1 or 4 output neurons.

is used in the case of multiple input patterns, with the output neurons inhibiting each other in a structure which resembles a soft Winner-Take-All network. All the synapses have a $1msec$ delay.

The input neuron layer comprises 800 neurons and is connected using plastic excitatory synapses to the output layer. The type of plasticity and the parameters of the network are described for each of the tests presented. The output layer is comprises tonic spiking Izhikevich neurons (i.e. their parameters are $a = 0.02$, $b = 0.2$, $c = -65$ and $d = 8$). The synapses present a first order dynamics (as described earlier in section 3.4.1) with a time constant different for excitatory and inhibitory synapses: $\tau_e = 1$ and $\tau_i = 20$.

An example of the input provided to the network is shown in Fig.6.9: the input layer is divided in two subclasses equally partitioned. The first subclass receives only noise while the second class receives one (or two) pattern(s) hidden in the noise. When there are two input patterns, they are interleaved between each other (1-2-1-2-1-...). The input noise is generated through a Poisson process with a mean firing rate of $25Hz$. The pattern to be identified is generated in the same way, and has the same firing rate as the noise, therefore it is not easily identifiable from an inspection of the raster plot of the input (see Fig.6.9). The inter-pattern time is itself a Poisson process with a mean modulation frequency of $9Hz$; patterns cannot, however, be present across the boundaries of a second, which slightly reduces the mean presence of the pattern from the planned $9Hz$.

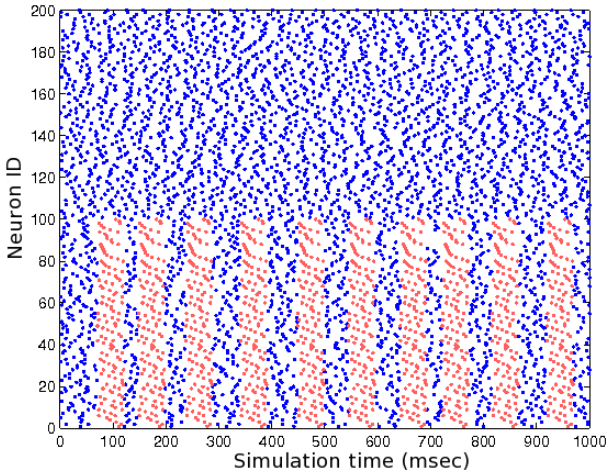


Figure 6.9: Example of raster plot of the input pattern for an input layer of 200 neurons and pattern sent to the first half of this population. The input pattern is highlighted in red. In the simulation the input was generated for 800 input neurons of which only half will receive the input pattern.

The outcome of all the tests are presented in the form of "scatter plots" (e.g. Fig.6.10(a)). In these graphs the horizontal axis represent the simulation time (in seconds); the vertical axis represents the time between the beginning of the pattern injection and the output spike (in milliseconds). Every dot represents an action potential emitted by (one of) the output neuron(s), as described in the figure caption. The spikes emitted within the time window of the input pattern (delimited by the red line in the graphs) are regarded as positive identification. The spikes above the line delimiting the pattern time window are considered false positives and therefore represent noise in the output. However, in some cases there is an identification which is just across the boundary line, and which remains consistent while the simulation advances. In this case, as will be explained in the text related to each of the images, this will be regarded as a positive identification of the pattern, even if slightly delayed form the end of the pattern itself.

6.7 Simulation results from SpiNNaker

Tests on this novel learning rule comprise a comparison with the standard STDP rule and the spike-pair STDP rule. In addition, a test without any learning rule is performed for completeness and subsequently some tests with the STDP TTS learning rule are run and presented. The discussion on the proposed results will highlight the extent to

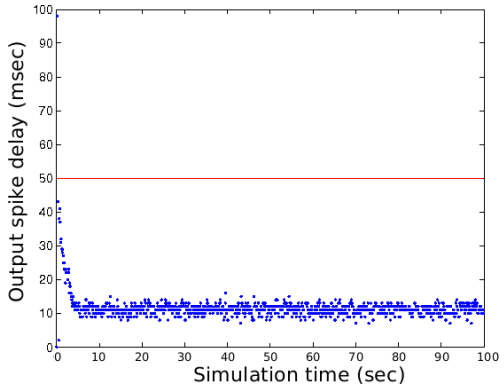
which the various learning rules provide similar results, and, therefore, accomplish this particular task satisfactorily.

The neural network used for this test is depicted in Fig.6.8(a). The initial weights of the synapses between the input and the output layers are set randomly in the interval $[0;0.4]mA$ with a uniform distribution, while the hard limits for the low and high bounds for the synaptic weights are set to the same interval.

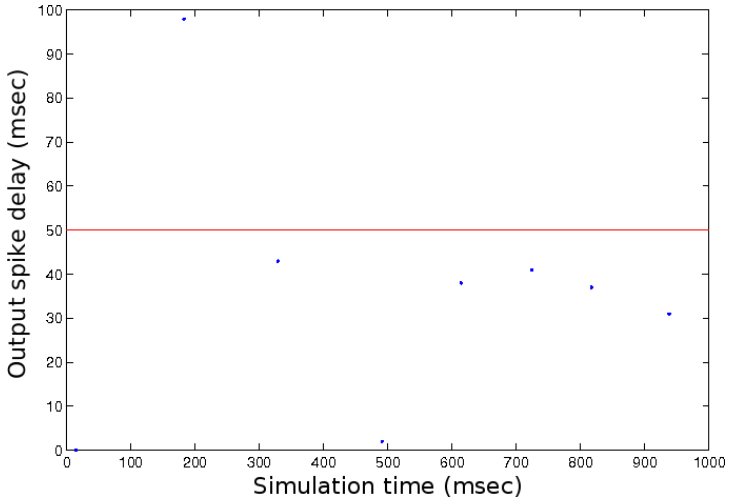
The STDP curve used in these experiments is composed by a function described by the two equations in (6.1), where the parameters used in the simulations are: $A_+ \approx 0.105$ and $A_- \approx 0.126$, $\tau_+ = \tau_- = 20$ (Song et al., 2000; Jin et al., 2009, 2010c).

Fig.6.10 presents the results of the simulation in terms of scatter plots, as described earlier, using the above parameters. The first of these plots (Fig.6.10(a)) presents the output of the network for a simulation using the standard STDP rule. This graph is divided in three parts, which represent the three stages of the neural learning process, and presented in Fig.6.10(a), whose first $1,000msec$ of simulation are expanded in Fig.6.10(b). In this specific case the initial 4 spike events in Fig.6.10(b), ($0 \sim 500msec$, represent the condition in which the output neuron is not aware of any injected pattern, and therefore fires according to the initial synaptic weights, which are set randomly. The initial random spike emission ceases when the output neuron starts detecting statistical regularities in the input, with the fifth spike emitted in Fig.6.10(b). When this condition is met, the second stage of the learning process takes place: the graph shows that the neuron is starting to detect statistical regularities in the input (the pattern in this case) and it is possible to see that the neuron is tracing these statistical regularities until the output spike is emitted after a minimum delay value from the beginning of the pattern injection, when it becomes constant. The third section is represented by the remaining part of the graph which shows that the output neuron locked to the pattern and is stable in its recognition. In this condition the output neuron emits an action potential after receiving the minimum set of inputs required, as described theoretically by Guyonneau et al. (2005). It is interesting to note that, using the described set of parameters, the simulation presents no false positive identification once it has locked to the pattern.

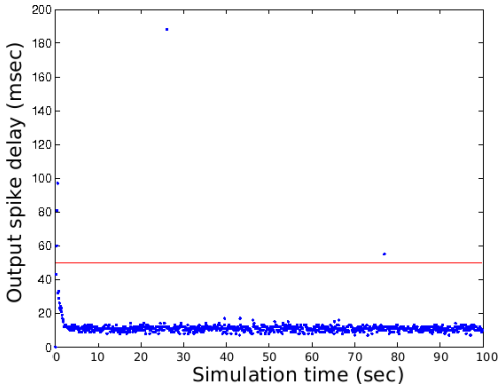
A similar set of events can be described when using the spike-pair STDP rule (Fig.6.10(c)). The first section on the left represents the output of an untrained neuron which tries to detect a pattern. Eventually it locks to it and tracks the pattern until it encounters the first group of spikes, as described for the standard STDP rule. In this case, the number of false positive identification of the input pattern is very low: only



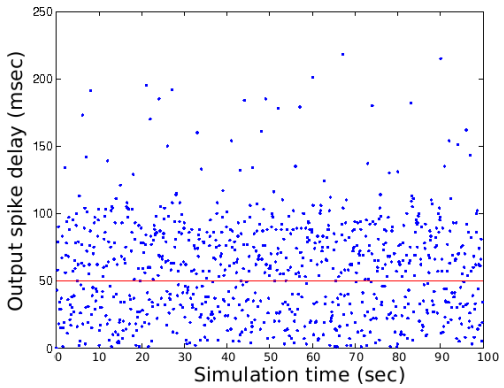
(a) Standard STDP rule



(b) Expansion of the first second of the scatter plot of (a)



(c) Spike-Pair STDP rule



(d) No learning rule used

Figure 6.10: Scatter plots for simulations using various learning rules.

two action potentials are emitted beyond the pattern limit, after the output neuron has locked on the pattern.

Finally, for completeness of this description, Fig.6.10(d) presents the output of the network in case that no learning rule is embedded in it. It is possible to see how the emission of action potentials by the output neuron is random with respect to the input pattern.

In the next section the output of the STDP TTS learning rule is presented. This rule requires one additional parameter, as described earlier: the "L" parameter. Various simulations for different values of this parameter are presented in the next section, and a particular value is defined on an experimental basis.

6.8 Manipulation of the "L" parameter

In this paragraph four series of tests using the STDP TTS learning rules are presented. Apart from the initial and the boundaries of synaptic weights, the parameters for all experiments are kept constant. The first set uses the same parameters and inputs used in the previous experiment to test the standard STDP and the spike-pair STDP rules:

- Network parameters:
  - Input neurons: 800;
  - Output neuron: 1;
  - Izhikevich neuron parameters: $a = 0.02$, $b = 0.2$, $c = -65$ and $d = 8$ (tonic spiking neurons);
  - Initial synaptic weight set randomly with a uniform distribution in the interval $[0;0.4]mA$
  - Synaptic weight limited in the interval $[0;0.4]mA$
- STDP curve parameters:
  - $A_+ \approx 0.105mA$;
  - $A_- \approx 0.126mA$;
  - $\tau_+ = \tau_- = 20msec$;
- Input parameters:

- Background noise frequency: $25Hz$;
- Pattern spike frequency: $25Hz$;
- Pattern repetition frequency in the background noise: $9Hz$;

The 8 scatter plots presented (first column in Fig.6.11 and Fig.6.12) differ from each other only in the "L" parameter used: the value assigned to it varies from $L = -63mV$ to $L = -70mV$.

The second set presented (second column Fig.6.11 and Fig.6.12) modifies the initial synaptic weights to be uniformly randomly chosen in the interval $[0; 1.5]mA$ and the limits of the synaptic weight are modified accordingly to the same interval.

The last two series (Fig.6.13 and Fig.6.14) have the same network configuration as the first and second sets, but the random number generator seed for the input and for the synaptic weight random number generators has been changed, with the purpose of testing the robustness of the STDP TTS learning rule.

6.9 Observations on the learning behaviour

From the scatter plots presented it is possible to note two main possible behaviours of the output neuron: the first one, observed initially by Guyonneau et al. (2005), shows that the learning algorithm lets the output neuron adapt to the incoming pattern and, since its membrane potential is kept above the threshold set by the "L" parameter (this threshold will be called later learning threshold, as the potentiation is triggered only when the post-synaptic neuron membrane potential is above such threshold), it is able to tune to the earliest possible group of spikes of the pattern. Example of this behaviour is presented in Figures 6.11(e) and 6.11(h).

The lower limit for the time required to identify the input pattern is set by the number of spikes (and indirectly by the input synaptic weights) required to bring the membrane potential above the learning threshold. However in some cases it is possible that the input pattern is recognised very early: $< 10msec$ after presenting the pattern. Fig.6.12(b, d, f and h) are examples of this behaviour. In this case it is possible that the neuron emits an action potential towards the end of the input pattern or even outside the boundary of the pattern. This behaviour shows that the neuron has learned two different sets of the input pattern and is tracking both of them at the same time, and both events are regarded as positive identification. However, as it is possible to note,

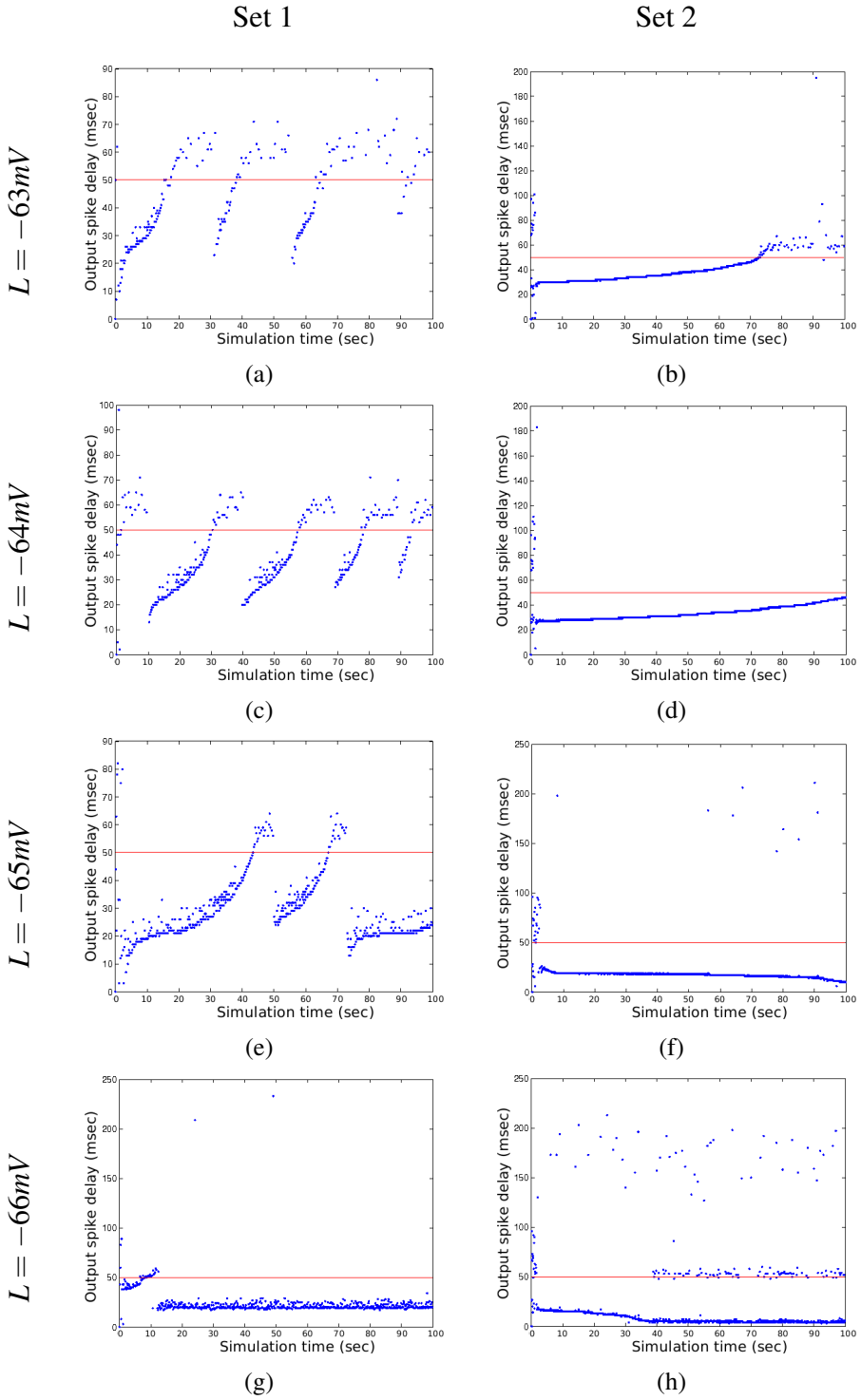


Figure 6.11: Scatter plots for STDP with TTS forecast - $-66mV \leq L \leq -63mV$. Set 1 has synaptic weights in the interval $[0;0.4]mA$. Set 2 has synaptic weights in the interval $[0;1.5]mA$.

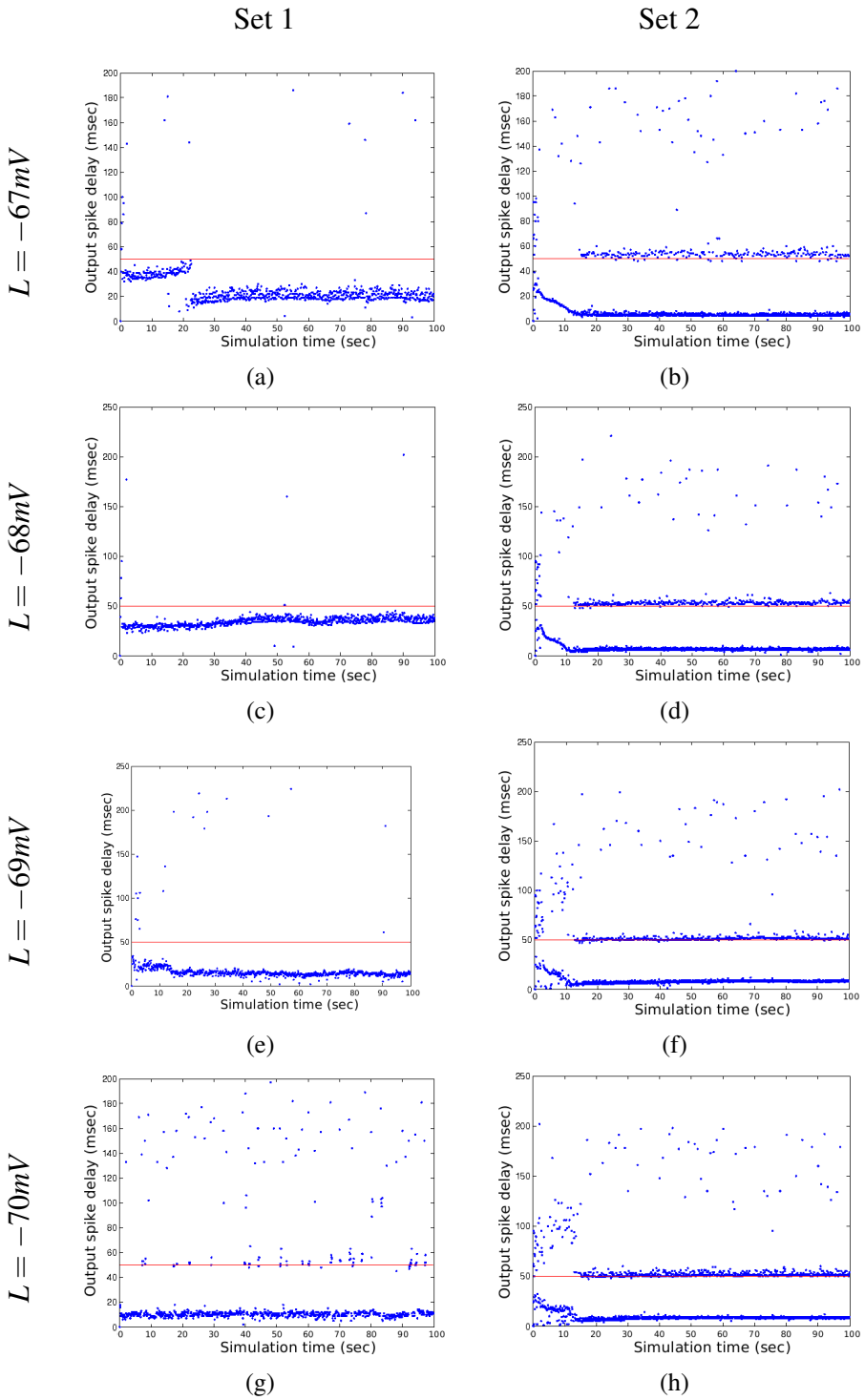


Figure 6.12: Scatter plots for STDP with TTS forecast - $-70mV \leq L \leq -67mV$. Set 1 has synaptic weights in the interval $[0;0.4]mA$. Set 2 has synaptic weights in the interval $[0;1.5]mA$.

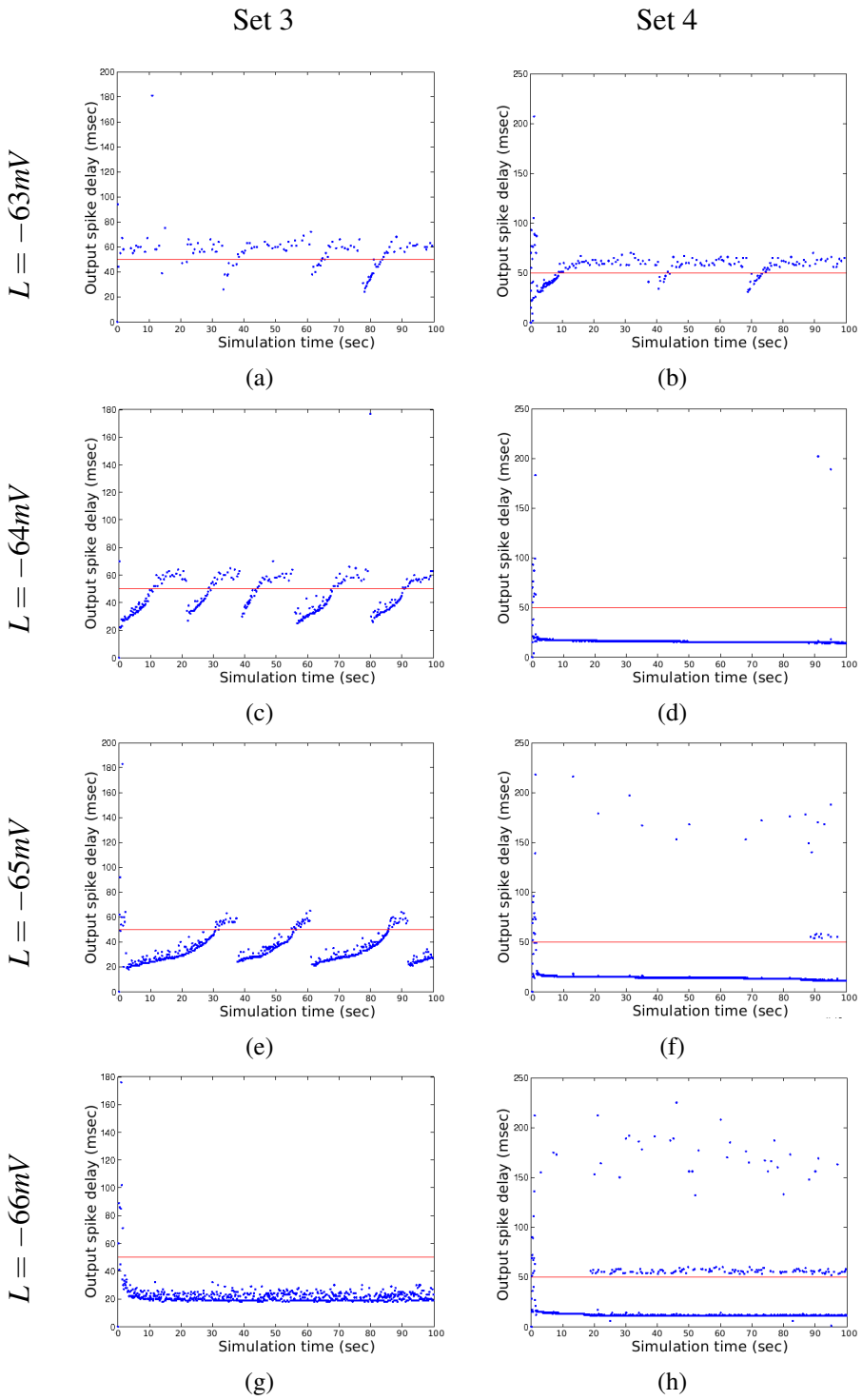


Figure 6.13: Scatter plots for STDP with TTS forecast - $-66mV \leq L \leq -63mV$. Set 3 has synaptic weights in the interval $[0;0.4]mA$. Set 4 has synaptic weights in the interval $[0;1.5]mA$. The seed of the random number generator used to generate the input and the initial synaptic weights for both sets of experiments has been changed from the experiments in Fig.6.11 and 6.12.

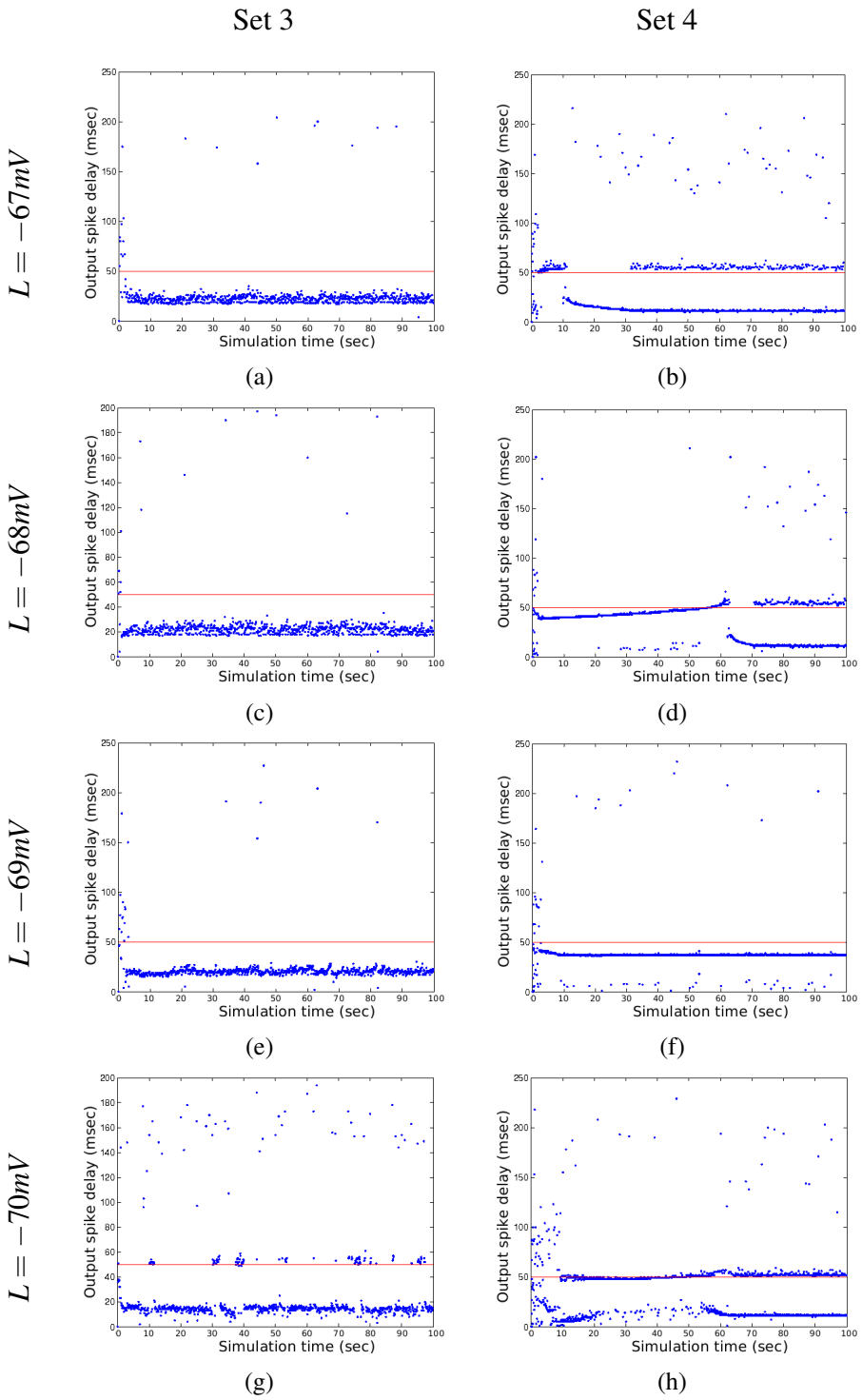


Figure 6.14: Scatter plots for STDP with TTS forecast - $-70mV \leq L \leq -66mV$. Set 3 has synaptic weights in the interval $[0;0.4]mA$. Set 4 has synaptic weights in the interval $[0;1.5]mA$. The seed of the random number generator used to generate the input and the initial synaptic weights for both sets of experiments has been changed from the experiments in Fig.6.11 and 6.12.

this condition generally presents a considerable amount of false positive identification of the input pattern (noise in the scatter plot).

The second possible behaviour of the output neuron is shown, for example, in Fig.6.11(e) and 6.11(d), and is the opposite of the behaviour STDP-like described earlier: once the output neuron locks to the input pattern, the identification happens later and later until the output neuron fires outside the pattern time window.

To understand this behaviour, an experiment utilising a simpler network is required. The network used belongs to the class described in Fig.6.8(a), but has only 100 input neurons and one output neuron. The input is generated with the same parameters used for the test with 800 input neurons, but this time 50 neurons receive only noise and the other 50 neurons receive the input pattern interleaved with noise. The initial synaptic weight is set to $4mA$ for all the synapses and the interval in which the synaptic weight can change is set to $[0;4]mA$.

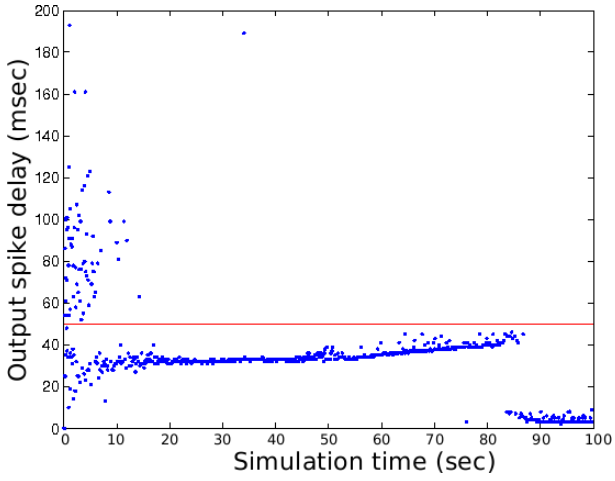


Figure 6.15: Scatter plot of a network with 100 input neurons using the STDP TTS learning rule.

In the scatter plot obtained from such test (Fig.6.15) it is possible to see that the output neuron fires later and later from the beginning of the pattern. For this simulation the evolution of each synaptic weight has also been recorded, and Fig.6.16 shows the evolution of the synaptic weight for the synapses which carry only noise. On the horizontal axis the time of simulation in seconds is shown; the vertical axis shows the synaptic weight value, limited to the interval $[0;4]mA$. Each of the coloured lines in the graph describes the evolution of the weight of each synapse; it is possible to see that all rapidly converge to $0mA$, as they do not carry statistical regularities in the injected

pattern.

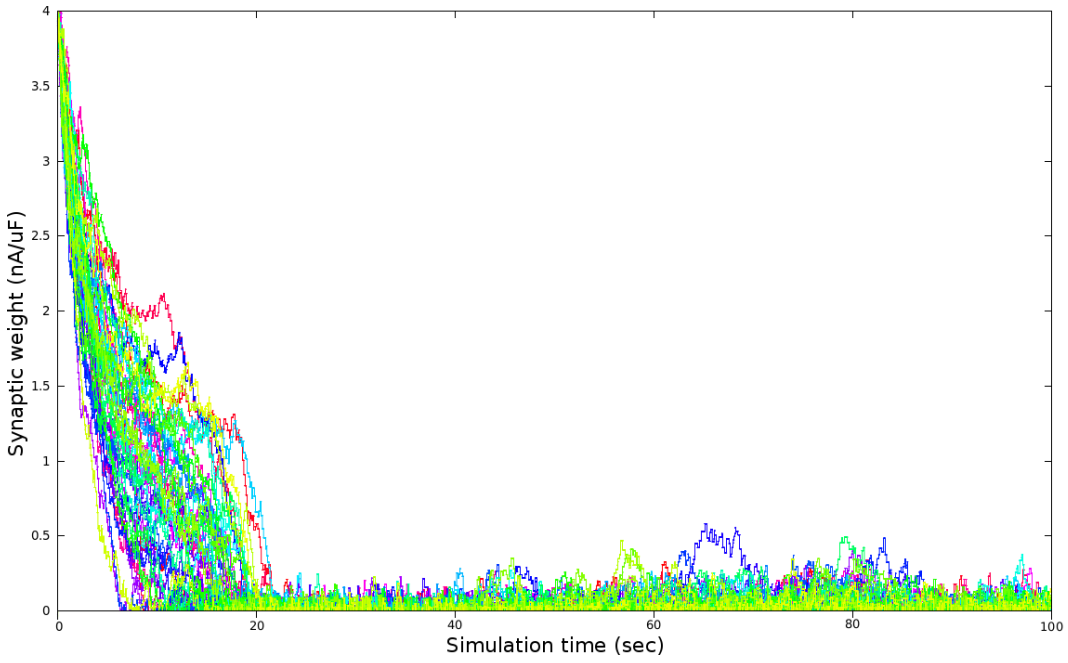


Figure 6.16: Evolution of the weights for the synapses that carry only noise.

Before presenting the evolution of the remaining set of synapses, these which carry the input pattern, it is important to analyse the input pattern itself (Fig.6.17). The input pattern is presented as a matrix where each row represents the input to each single neuron, and each column represents the input injected in each millisecond for all the neurons. A "1" in the matrix identifies a spike for a specific neuron in a specific millisecond. In addition, the last row identifies the number of spikes which the output neuron receives in each millisecond of the pattern. The last column marks the neurons which send multiple input spikes in a single pattern.

From the scatter plot in Fig.6.15 it is possible to see that the pattern identification locks after about 20*sec* of simulation at $\approx 35msec$ from the beginning of the pattern. For this reason, looking at the plot of the evolution of the synaptic weight for one of the early spikes of the pattern (e.g. input neuron 22, millisecond 12 of the pattern, Fig.6.18), it is possible to see how the synaptic weight decreases. From this result, it is possible to deduce that, at the time the spikes from these synapses are received, the output neuron is insufficiently "charged" (i.e. its membrane potential is not enough high) to trigger potentiation in the STDP TTS algorithm. In the description of these events, and for this discussion, the first 80 seconds of the simulation are considered.

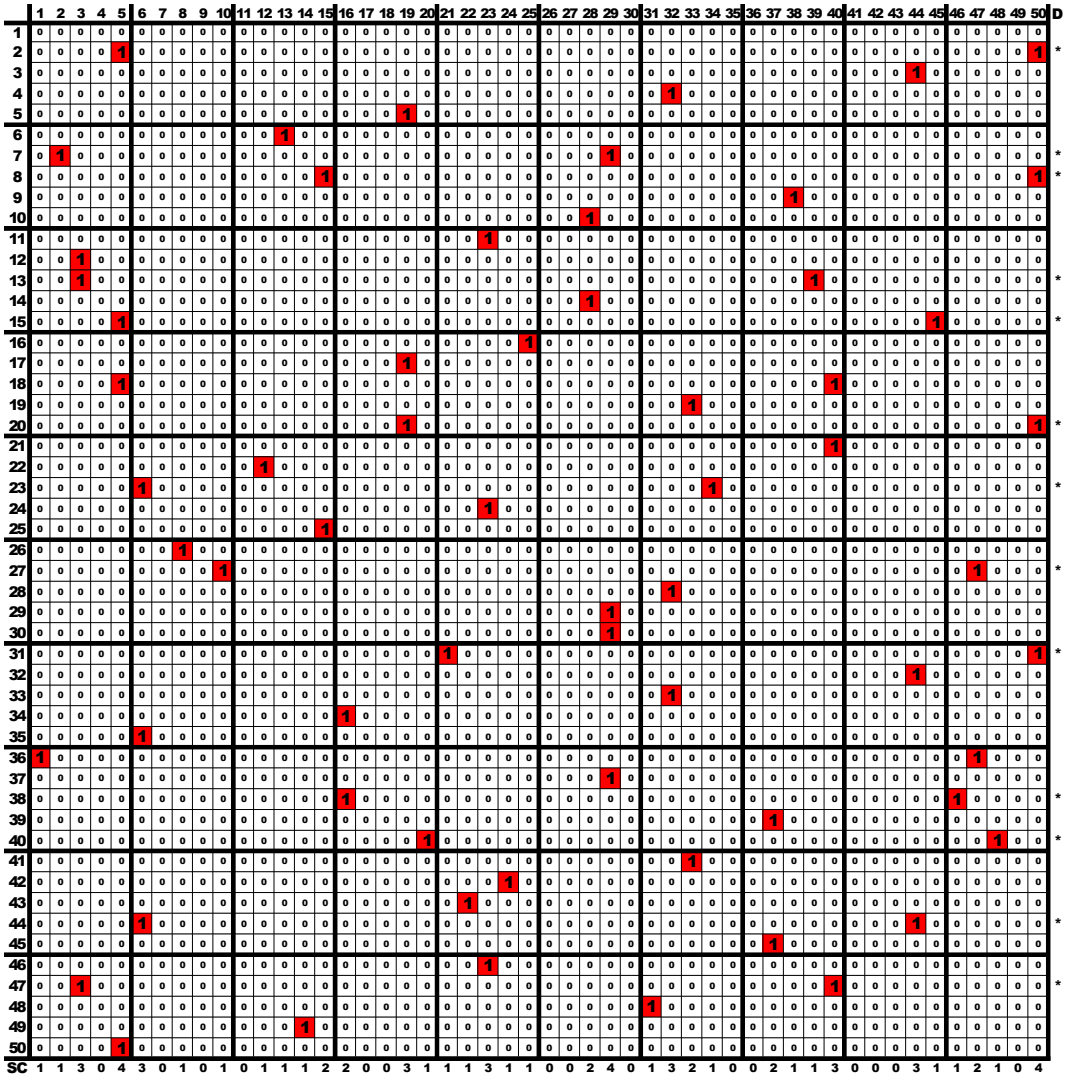


Figure 6.17: Representation of the input pattern.
Each row represents the input to each single neuron. Each column represents the input injected in each millisecond for all the neurons. A "1" in the matrix (highlighted by a red cell) identifies a spike emitted by a specific input neuron in a specific millisecond. The last row identifies the number of spikes which the output neuron receives in each millisecond. The last column marks the neurons which send multiple input spikes in a single pattern.

The remaining part is discussed subsequently, towards the end of this paragraph, as the behaviour is the result of a peculiar process which needs to be analysed in-depth.

As synapses carrying spikes which are more advanced in the pattern (i.e. that carry spikes which are injected later in the pattern) are considered, the evolution of the synaptic weights involves higher peak values, because the membrane potential of the output neuron has been raised by previous spikes. The synapse related to input neuron 34 (Fig.6.19), active at millisecond 16 of the input pattern has a maximum value of $\approx 2.75mA$ around 19 seconds from the beginning of the simulation; it then starts decaying again because the weight of the previous synapses has also decreased. Therefore the output neuron's membrane potential is no longer above the learning threshold, and consequently the synapse under analysis cannot be potentiated.

Considering the synapses active during millisecond 23 of the input pattern (Fig.6.20), it is easy to note that the maximum peak is noticeably lower than the one previously seen: in fact the peak value is $< 2.5mA$ after approximately 19 seconds of simulation. This apparent anomaly can be justified considering the number of spikes in the previous milliseconds of the pattern: in the period between milliseconds 20 and 22 of the input pattern only 1 spike per millisecond is received by the output neuron from the relevant synapses (the synapses that carry the pattern). This causes the membrane potential of the output neuron to decrease by more than the contribution coming from the excitatory synapses, hence the potentiation of the three synapses analysed is not particularly strong. However, since these three synapses have a strong contribution to the membrane potential, the synapse related to the spike of millisecond 24 of the pattern (neuron 42, Fig.6.21) is strongly enhanced for a longer period of time, and reaches the saturation value for some seconds.

However, as the weight of previous synapses decreases, the membrane potential of the output neuron is no longer above the learning threshold, and therefore LTP cannot be triggered for this synapse. For a short transitory period of the simulation, the contribution of this synapse is still enough strong to push the output neuron's membrane potential above the learning threshold, so that the subsequent synapse continues to be potentiated for a short while longer (i.e. input neuron 16, millisecond 25 of the pattern, Fig.6.22).

This is a cascade effect which repeats for every synapse from this point in the pattern onward, and as a result, the output neuron delays the emission of the action potential further because it requires more time to bring the membrane potential over the firing threshold. Therefore the output neuron has the effect of "escaping" from the

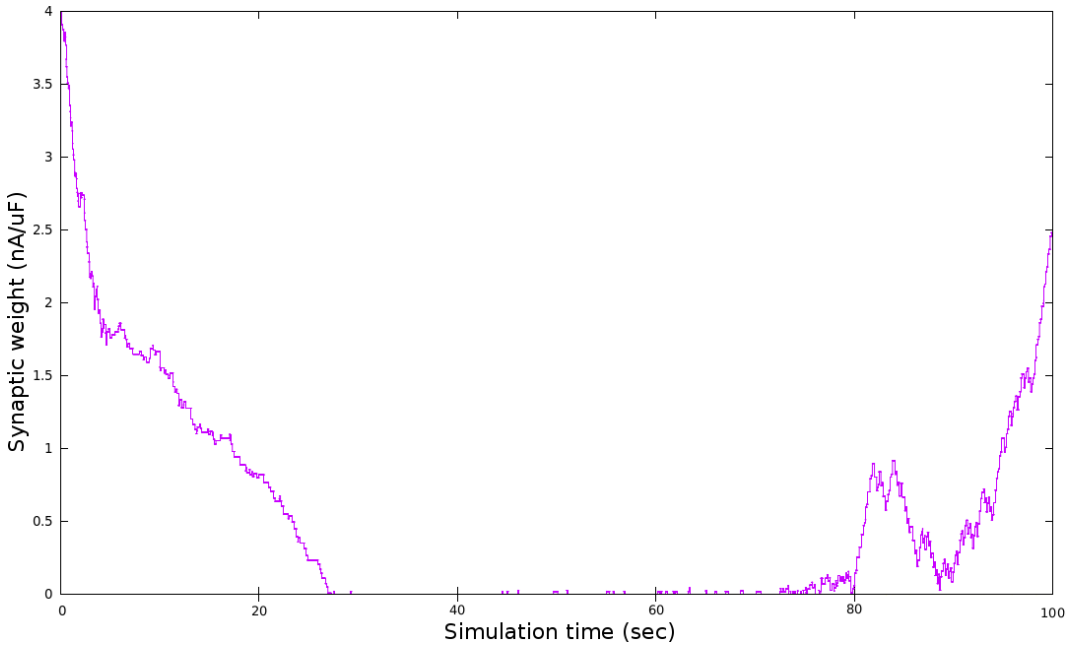


Figure 6.18: Evolution of the weights for the synapse of input neuron 22, millisecond 12 of the pattern.

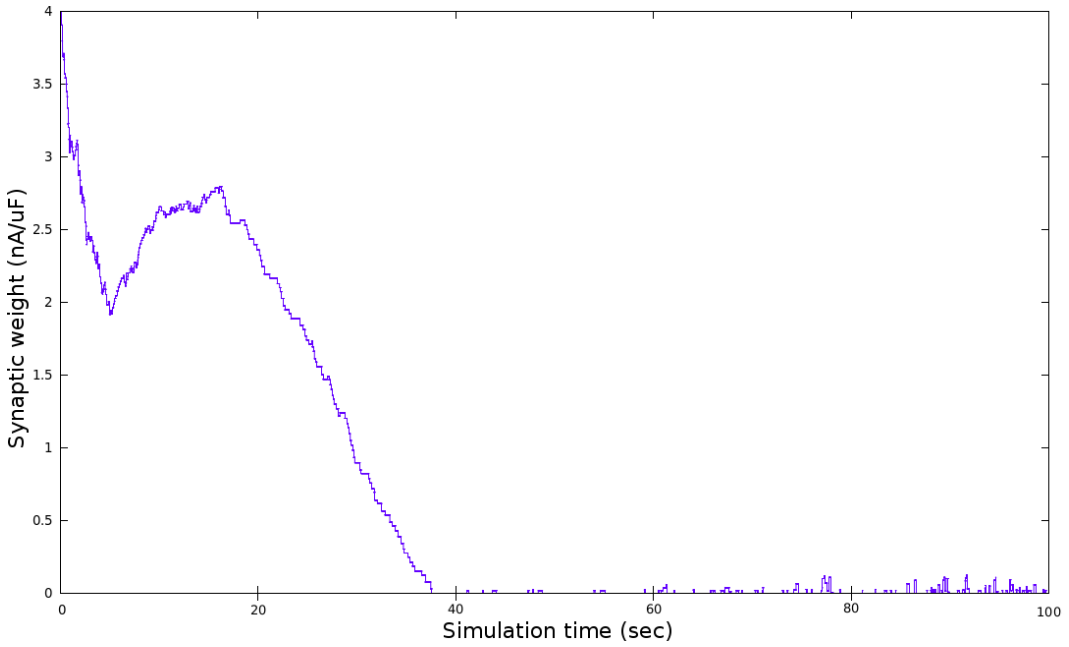


Figure 6.19: Evolution of the weights for the synapse of input neuron 34, millisecond 16 of the pattern.

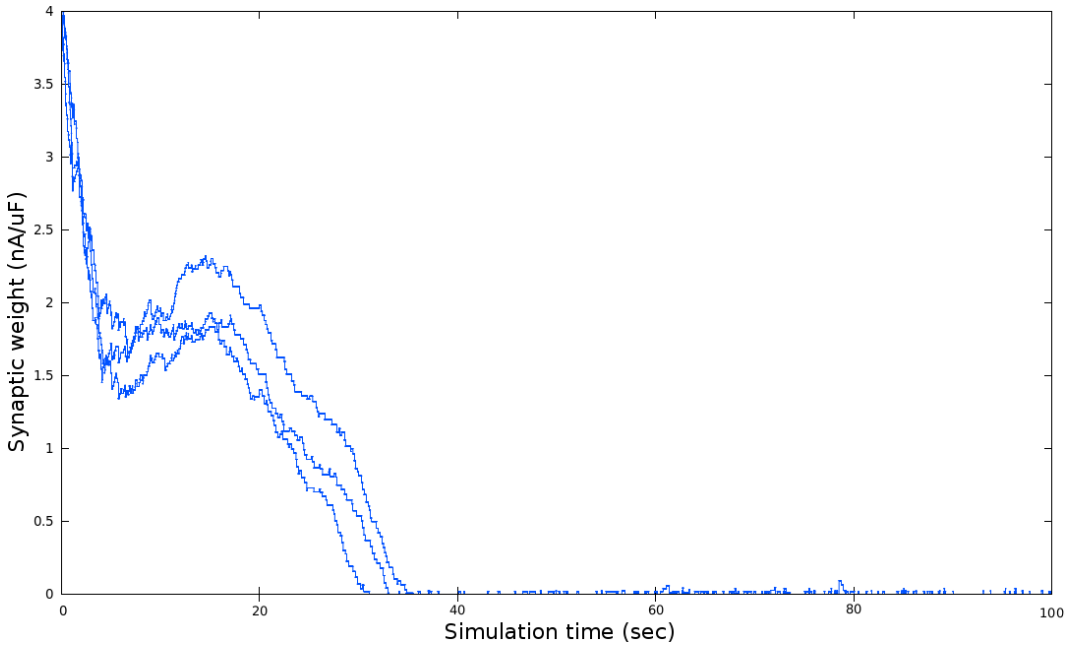


Figure 6.20: Evolution of the weights for the synapse of input neurons 11, 24 and 46, millisecond 23 of the pattern.

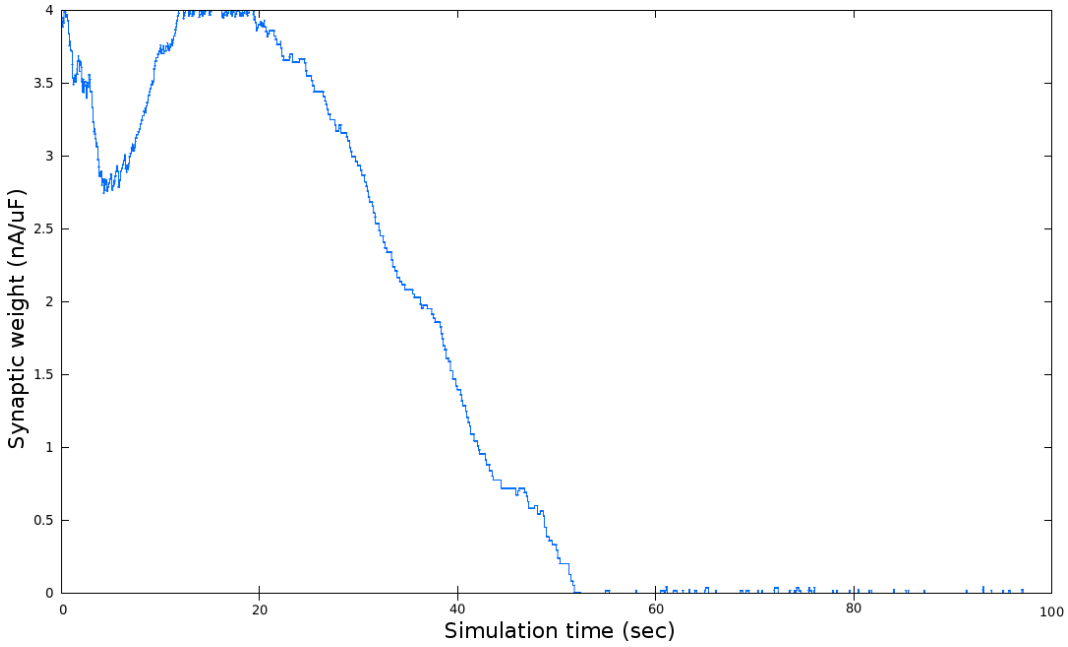


Figure 6.21: Evolution of the weights for the synapse of input neuron 42, millisecond 24 of the pattern.

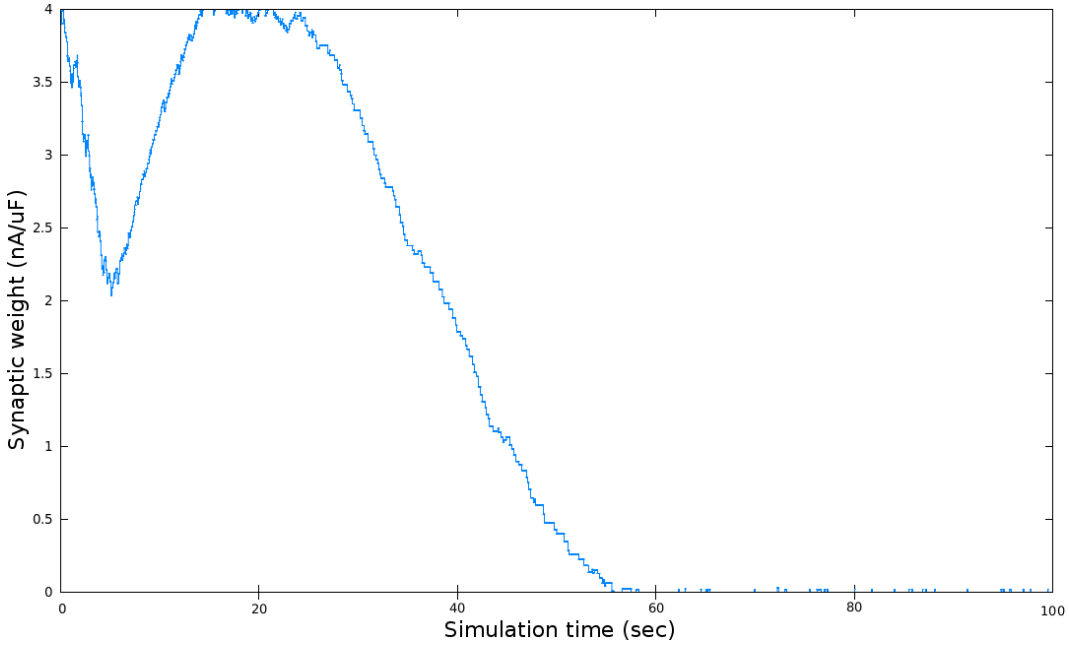


Figure 6.22: Evolution of the weights for the synapse of input neuron 16, millisecond 25 of the pattern.

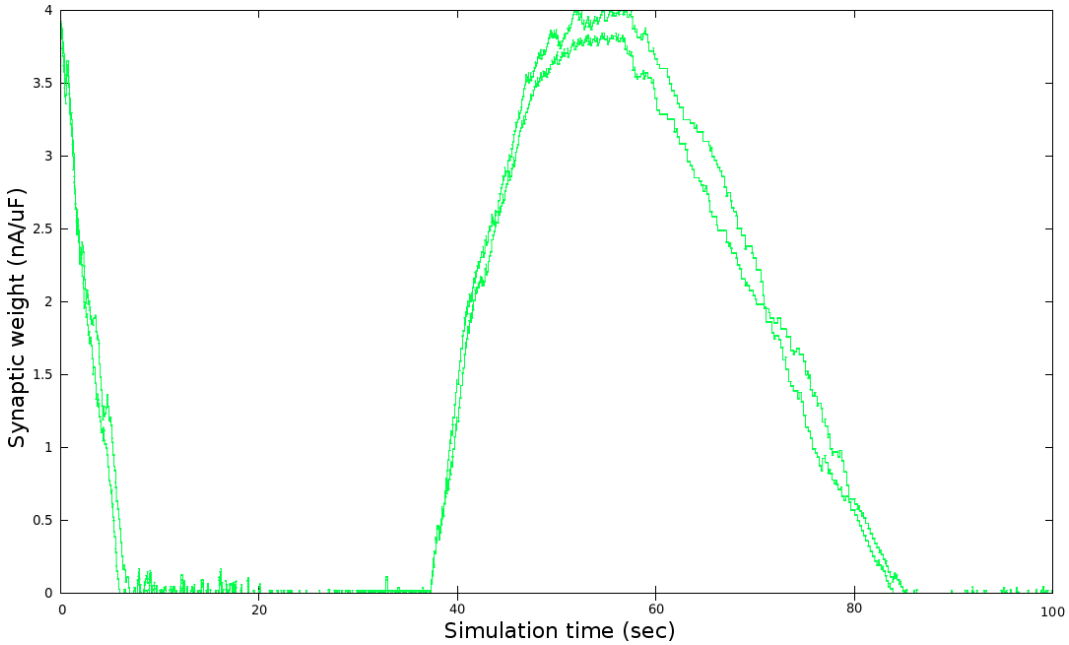


Figure 6.23: Evolution of the weights for the synapse of input neurons 39 and 45, millisecond 37 of the pattern.

time window of the input pattern.

An interesting behaviour is presented by the synapses of neurons 39 and 45, which are active at millisecond 37 of the input pattern (Fig.6.23). As can be seen, the weight of the synapses is $\approx 0mA$ for about 38 seconds of simulation (except the initial transitory state). This happens because, initially, the input from these neurons is received after the output neuron has fired, and therefore the membrane potential is close to the reset state, below the learning threshold. However, after 38 seconds of simulation, the emission of the action potential has been delayed until the point when the membrane potential of the output neuron, at this point of the pattern, is above the learning threshold. After a short while (about 20 seconds of simulation), the synapses cannot get more potentiation because the weight of the previous synapses has decreased to the point where their contribution does not push the membrane potential above the learning threshold, and therefore the synapses of neurons 39 and 45 are themselves depressed.

The description of the events done so far can lead to various evolutions of the experiment. In the scatter plot of the last example (Fig.6.15) it is possible to see that after ≈ 85 seconds of simulation the identification jumps from $\approx 40msec$ to $\approx 10msec$ of delay since the beginning of the pattern.

This behaviour is one of the possible evolutions, and this case depends on the structure of the input pattern: from the input pattern in Fig.6.17 it is possible to see that several neurons fire twice in the pattern, the first time at the beginning and the second towards the end of the pattern. This causes a "wrap-around" effect: as the weight of these synapses increases because the detection of the pattern is delayed towards the end of the pattern, the contribution given by the same synapses at the beginning of the pattern increases as well. Thus the detection moves to the beginning of the pattern, and this causes the weights of some of the other synapses to start increasing again after 80 seconds of simulation.

A different evolution of the events is represented (with some approximation) in Fig.6.13(b): the detection of the input pattern happens consistently and (more or less) permanently after the limit of the input pattern time window. However, in all the graphs which show the "escaping" behaviour, it is possible to note the (almost complete) absence of false positive identifications of the input pattern.

The scatter plots presented earlier (Fig.6.11, Fig.6.12, Fig.6.13 and Fig.6.14) show that a learning behaviour, similar to the one presented by the standard STDP (Fig.6.10(a)) and the spike-pair learning rule (Fig.6.10(c)), is achievable with a "L" parameter included in the interval $-66mV \leq L \leq -64mV$: (see Fig.6.11(g), Fig.6.11(f),

Fig.6.13(g), Fig.6.13(d)), depending on the interval set for the input synaptic weight, and therefore on the mean value of the membrane potential of the output neuron.

6.10 Influence of the parameters

This section presents how some of the parameters of the network influence the learning behaviour of the STDP TTS rule. In the following paragraphs some of the parameters of the networks introduced before are changed with the purpose of analysing the effects and discussing the results.

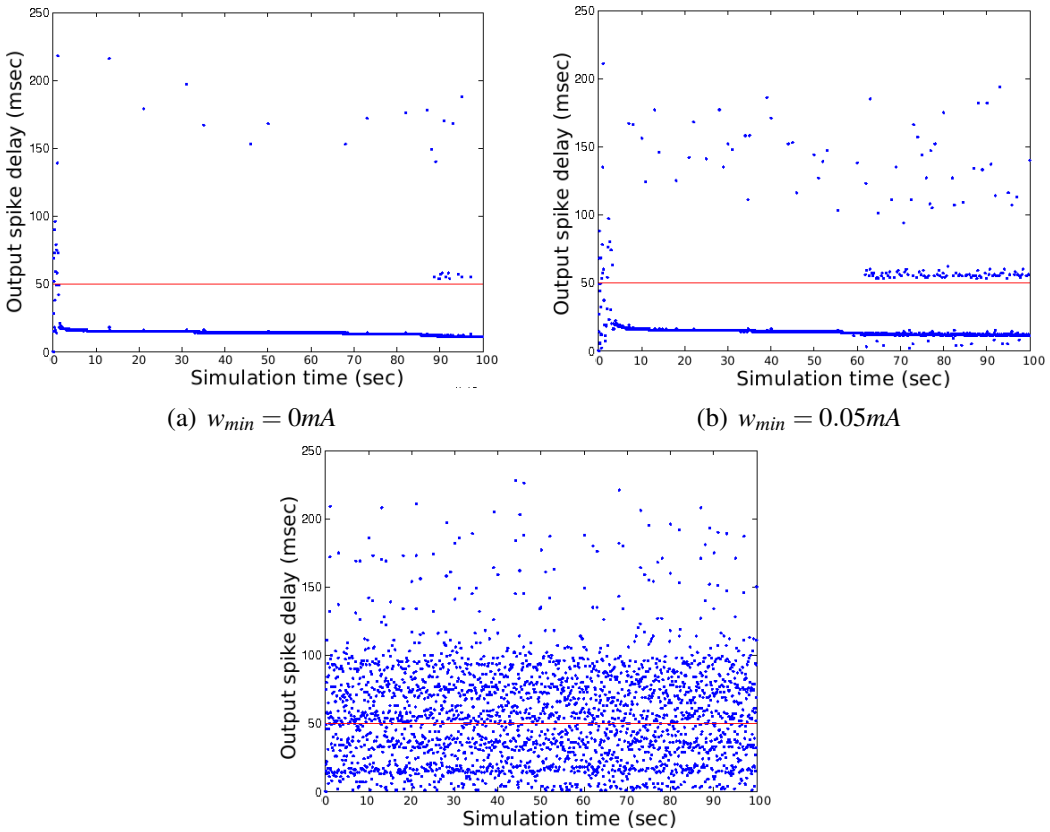
6.10.1 Increasing maximum synaptic weight

This behaviour has already been shown in the comparison between sets 1 and 2 expressed in section 6.8: it is possible to see that the resulting scatter plot in set 1 (Fig.6.11(g)) is slightly more noisy that the scatter plot in set 2 (Fig.6.11(f)). This is justified on the basis that the synaptic weight modification related both to LTP and LTD keeps a value which is independent of the interval of the synaptic weight. In the case of set 1, the maximum allowed synaptic weight is $0.4mA$, while the synaptic weight modification values are $\Delta w_{LTP} = 0.1mA$ and $\Delta w_{LTD} = 0.12mA$, representing a significant percentage of the maximum allowed synaptic weight ($\approx 25\%$). On the other hand, for set 2, the maximum allowed synaptic weight is $1.5mA$, and in this case the maximum synaptic weight modification is, in percentage, relatively smaller: $\approx 6.7\%$.

Additionally, since the synaptic weight in set 2 is stronger than in set 1, the membrane potential of the output neuron is generally at a higher level in set 2, therefore the learning threshold must be set accordingly to observe a similar behaviour.

6.10.2 Incrementing minimum synaptic weight

Incrementing the minimum synaptic weight has two main effects: firstly, as the weight of the synapses cannot any more reach 0, the mean value of the membrane potential of the output neuron increases. This effect facilitates learning as the output neuron's membrane potential is more likely to be above the learning threshold. The second effect is that the output neuron is no longer able to select efficiently which synapses carry statistical regularities. The first effect allows the identification of the input pattern to advance further toward the beginning of the pattern, according to Guyonneau et al. (2005). The second effect tends to increment the noise in the scatter plot; this



(a) $w_{min} = 0mA$ (b) $w_{min} = 0.05mA$

(c) $w_{min} = 0.5mA$

Figure 6.24: Scatter plot using the STDP TTS learning algorithm changing the minimum allowed synaptic weight

is shown in the comparison between Fig.6.24(a) (which is the same as Fig.6.13(f)), and Fig.6.24(b), obtained using $W_{min} = 0.05mA$. The two scatter plots are extracted from a simulation where the parameters are: initial synaptic weights are defined randomly in the interval $[W_{min}; 1.5]mA$; the synaptic weight value is defined in the interval $[W_{min}; 1.5]mA$; with regards to the STDP curve, the parameters are: $A_+ \approx 0.105$ and $A_- \approx 0.126$, $\tau_+ = \tau_- = 20$, and the "L" parameter is set to $L = -65mV$. The network uses 800 input neurons and 1 output neuron, and the input provided is the same as in the previous test.

However, if the minimum synaptic weight has a too high value, the output neuron is not able to select which synapses carry relevant information and which do not, as all will have a significant contribution to the membrane potential (e.g. $W_{min} = 0.5mA$, Fig.6.24(c)).

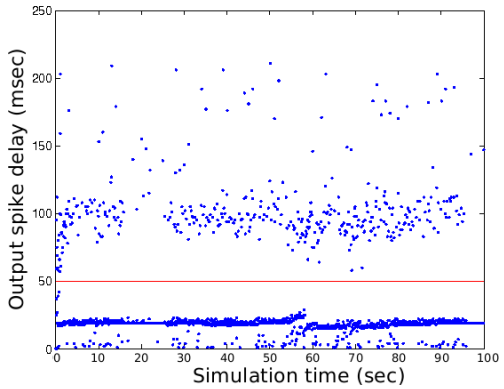


Figure 6.25: Scatter plot using a learning rule with the STDP curve that uses $A_+ = 0.3$, while the other parameters are set as before.

6.10.3 Incrementing LTP amplitude parameter

The last parameter modification which is analysed in the characterization of this novel learning rule is the parameter A_+ of the STDP curve. Previously, this value has been set to $A_+ = 0.106$; for this experiment it has been raised to $A_+ = 0.3$, and the result is shown in the scatter plot in Fig.6.25.

This parameter influences the ability to adapt to possible patterns which are revealed in the input, therefore the output neuron is more inclined to detect a new pattern, even when there is none. As a consequence, the noise in the output is greatly increased and, as a consequence of this, since the spikes are very close to each other, the synapses carrying the real pattern are depressed through the LTD mechanism. The final consequence of these events is that the output neuron presents a slight "escaping" behaviour, as described before in section 6.9.

6.11 Multiple input patterns

A final test on the learning behaviour of the STDP TTS learning rule involved a more complex environment in which the network in Fig.6.8(b) was stimulated with two input patterns immersed in noise. The configuration of the network used the following parameters:

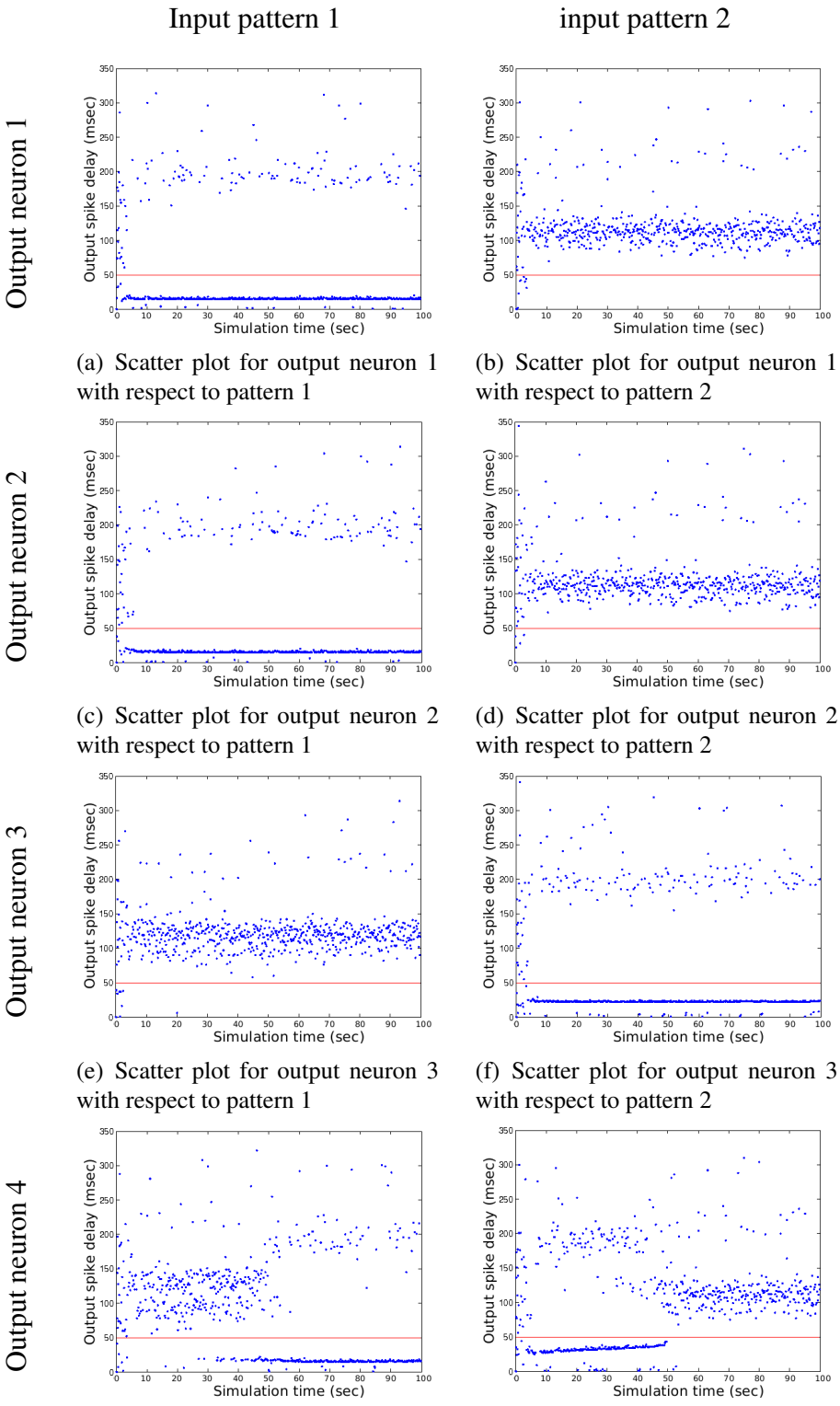
- Input neuron layer composed by 800 neurons;
- Output neuron layer composed by 4 neurons;

- The output neurons inhibit each other (but they do not self-inhibit);
- Initial synaptic weight for input synapses uniformly distributed in the interval $[0;1.5]mA$;
- Synaptic weight boundary limits: $[0;1.5]mA$;
- Learning parameter $L = -65mV$;
- Parameters of the STDP curve: $\tau_+ = \tau_- = 20$, $A_+ \approx 0.105$ and $A_- \approx 0.126$;
- Time constant for first order excitatory synapses: $\tau_E = 1.0$
- Time constant for first order inhibitory synapses: $\tau_I = 20.0$
- Synaptic weight for inhibitory synapses between the output neurons $5mA$;

The input has been generated with the same parameters described earlier: the input noise and the patterns to be identified are generated through a Poisson process with a mean firing rate of $25Hz$. The inter-pattern time is itself a Poisson process with a mean modulation frequency of $9Hz$. In this case the two patterns are interleaved with each other, so that the sequence is (1-2-1-2-1-$\cdots$). The outcome of this test is presented in Fig.6.26 which comprises 8 scatter plots, one for each neuron with respect to the timing of each of the input patterns.

As the initial synaptic weights are random, it is not predefined to which pattern and in particular to which section of the pattern each output neuron locks. Nonetheless each neuron acts with lateral inhibition pulses to prevent other neurons locking on the same pattern, However, because lateral inhibition acts with a one millisecond delay from an output spike, two output neurons firing simultaneously will mutually inhibit one millisecond after the initial spike, but they can identify the same millisecond of the pattern. Therefore it is possible that two output neurons (1 and 2 in this case) can lock at the same point of the first pattern (Fig.6.26(a) and 6.26(c)).

Output neuron 3 locks on pattern 2, and through lateral inhibition prevents output neuron 4 from keeping a lock on the same pattern (Fig.6.26(h)). Here, in fact, it is possible to see the "escaping" behaviour of the neuron which is triggered by the inhibition pulses coming from output neuron 3. When the identification moves toward the end of pattern 2, the output neuron locks to pattern 1 (Fig.6.26(g), in the very same moment as output neurons 1 and 2.



(a) Scatter plot for output neuron 1 with respect to pattern 1

(b) Scatter plot for output neuron 1 with respect to pattern 2

(c) Scatter plot for output neuron 2 with respect to pattern 1

(d) Scatter plot for output neuron 2 with respect to pattern 2

(e) Scatter plot for output neuron 3 with respect to pattern 1

(f) Scatter plot for output neuron 3 with respect to pattern 2

(g) Scatter plot for output neuron 4 with respect to pattern 1

(h) Scatter plot for output neuron 4 with respect to pattern 2

Figure 6.26: Scatter plots for STDP with TTS forecast - two input patterns, four output neurons.

As can be observed, in Figures 6.26(b), 6.26(d), 6.26(e) and partially in Figures 6.26(g) and 6.26(h), there is a "ghost" effect. This is connected with the presence of two different interleaved input patterns, whose inter-time is random. When a neuron identifies one of the two input patterns it fires with a delay from the beginning of the pattern identified and with a different delay (usually greater) from the beginning of the previous pattern occurrence. Given the randomness of time occurrence of the pattern, the result is the "ghost" effect shown before.

6.12 Performance evaluation

The performance of the learning algorithms developed for the SpiNNaker system, the standard STDP, the spike-pair rule and the STDP TTS rule, have been evaluated in terms of computational cycles required to perform and the memory occupancy of each of them.

6.12.1 Computational requirements

The tests run on the SpiNNaker hardware allowed measurement of the computational effort required by the processor to perform each of the learning algorithms (Tab.6.1). Since some of them can have a wide variability due to the input and history of the spikes, a minimum and a maximum case are considered, together with a usual case. The minimum case is measured for the standard STDP in the condition where no spikes have been received or emitted, so that all records are blank and no synaptic update is effectively triggered. The maximum case is measured for the standard STDP in the condition where, every millisecond, a spike has been received and emitted, so that all records show the full set of incoming and outgoing spikes. The last two tests are unrealistic from a biological point of view, but define the limits in terms of computational power required. For a more general case, a measurement of the cycles required for the learning algorithm during one of the experiments discussed earlier is taken. For the spike-pair STDP, the minimum case conditions are the same as the ones for the standard STDP algorithm; the worse case scenario, however, happens when a spike that is received and a post-synaptic spike that is emitted are recorded at the beginning and at the end of the record time window, requiring therefore an intensive search. The usual case is extracted, as described before, from the simulation run in one of the tests

	Min computation values		Mean computation values		Max computation values	
	Clock cycles	Time (μsec)	Clock cycles	Time (μsec)	Clock cycles	Time (μsec)
Standard STDP	425	2.125	1,708	8.54	365,424	1,827.12
Spike-pair STDP	386	1.93	1,021	5.105	1,282	6.41
STDP TTS	N.A.	N.A.	270	1.35	N.A.	N.A.

Table 6.1: Computation requirements for each learning algorithm (values are approximated).

described previously. For the STDP TTS algorithm, there is no best or worst case scenario, as the algorithm always performs the same steps with few and small variation in their sequence. Therefore, in this case, only the usual amount of computation will be presented.

It should be noted how all the learning rules, here described, are much more expensive, in terms of computational effort, compared with the requirements for the neural membrane update. For example, for an Izhikevich neuron, currently the most complex neuron simulated on the SpiNNaker architecture, Izhikevich in 2004 estimated the number of floating point operations (FLOPS) to be 13 (Izhikevich, 2004). This number is at least one order of magnitude lower than the computational complexity of the simplest learning rule implemented on SpiNNaker (see Tab.6.1).

In the following paragraphs, the computational requirements for each of the (macro) steps of the algorithm is presented.

Standard STDP

The standard STDP rule implementation comprises two main parts: the first is fixed, and is independent of the number of synapses which depend on the particular pre-synaptic neuron; to perform this part of the algorithm, ≈ 50 clock cycles are required. The second part of the algorithm, instead, is executed for each neuron to which a received spike has to be propagated. The initial operation of this group is to decode the synaptic word fetched from the external memory, and this task requires ≈ 32 clock cycles to complete. Measuring the computation required for the search of spike couples (one pre-synaptic and one post-synaptic spike) and the relative weight update for the

standard STDP algorithm leads to results which are not precise, as the number of operation varies with the history of the incoming and outgoing spikes. However, a mean value for such an operation may be extracted, and it is $\approx 1,582$ clock cycles. In this part of the algorithm, once the time stamp of the pre- and post-synaptic spikes have been extracted from the record, the routine which updates the weight takes ≈ 61 clock cycles to perform. Finally, the synaptic word needs to be encoded again before being stored back in the SDRAM; the number of clock cycles required for this task is very similar to that for the decoding task and requires ≈ 37 clock cycles.

Spike-pair STDP

This algorithm shares a relevant part of the code with the standard STDP rule, such as the setup part of the STDP algorithm and the synaptic weight update routine. The part which changes from the standard STDP rule involves the search for the two couples of spikes on which trigger LTP and LTD. In this case 676 clock cycles was the mean value required for the computation of the LTP and the LTD.

STDP TTS

The STDP TTS rule implementation includes two parts, as for all the other algorithms: a fixed, or setup, part and a repeating part. However, differing from the other algorithms, this one comprises a series of steps which do not vary greatly with the history of spikes received and emitted. In fact, there is no history kept of incoming spikes and the time stamp of only one outgoing spike is stored. The fixed part of the algorithm requires ≈ 22 clock cycles and the repeating part requires ≈ 260 clock cycles which can be approximately divided so:

- Word decode: 32 clock cycles;
- Voltage retrieval: 30 clock cycles;
- Forecast: 51 clock cycles;
- Synaptic weight update (LTP): 57 clock cycles;
- Synaptic weight update (LTD): 55 clock cycles;
- words encode: 37 clock cycles;

In conclusion, it is clear that the STDP TTS requires much less computation to perform the learning task compared with the other two learning rules. Additionally, the variability in the number of clock cycles is reduced.

6.12.2 Memory occupation

This analysis considered only the memory occupied by the data related to the learning rule for each neuron, plus any data used, in general, for each of the rule under comparison. No temporary data required for the computation of the algorithms is considered.

Standard STDP and spike-pair STDP

This rule takes into account the timing of all the incoming and outgoing spikes, so the memory is occupied by the record of the timestamps connected with these events.

- Memory related to incoming synapses: 2 words (8 bytes) for each pre-synaptic neuron. If the same pre-synaptic neuron is connected to multiple post-synaptic neurons, this data still occupies 2 words;
- Memory related to outgoing spikes: 5 words (20 bytes) for each neuron simulated in the core;
- Memory related to the STDP table: 257 bytes which contain the STDP function table and additional 9 bytes of information to configure the synaptic update routine.

The values related to the memory required to store the time stamp of incoming and outgoing spikes are defined in software and, therefore, are modifiable. However, the smaller the memory available for this purpose, the greater are the chances that some synaptic updates are not performed, and therefore the performance of the algorithm degrades.

STDP TTS

This rule needs to store only one post-synaptic spike which happens before a pre-synaptic spike is received. The space required is, therefore, connected with a single time-stamp record.

- Memory related to incoming synapses: 0 words;

	Standard STDP and Spike-Pair STDP	STDP TTS
Memory occupation for incoming synapse	2 words × 10,000 pre-synaptic neurons = 20,000 words = 80,000 bytes	0 bytes
Memory occupation for outgoing spikes	5 words × 100 neurons = 500 words = 2,000 bytes	1 word × 100 neurons = 100 words = 400 bytes
Memory related to STDP table and parameters	266 bytes	268 bytes
Total	82,266 bytes	668 bytes

Table 6.2: Example of memory requirements for each learning algorithm in the case of 100 neurons each receiving input from other 100 neurons.

- Memory related to outgoing spikes: 1 words (4 bytes) for each neuron simulated in the core.
- Memory related to the STDP table: 257 bytes which contain the STDP function table and additional 11 bytes of information to configure the synaptic update routine.

Some memory occupation figures can be extracted from an example deploying a hundred neurons, each receiveing spikes from a hundred different neurons. For this simple example the memory occupation is detailed in Table 6.2.

Memory occupation is, therefore, greatly reduced (by $\approx 99.1\%$) when comparing the STDP TTS with the standard STDP and the spike-pair rules.

6.13 Discussion

The STDP TTS algorithm permits aggressive improvements in computational and memory efficiency. However, the time required for the output neuron to fire to signal that a pattern injection has been identified, the *network* response time, may be slower because the output neuron is not able to tune to the earliest group of spikes of the input pattern(s) (Fig.6.7). A weak winner-take-all configuration with mutual inhibition in the four-output-neuron simulations is used to discourage two neurons from identifying the same portion of the input pattern. However, in some cases (Fig.6.26(a) and Fig.6.26(c)) the inhibitory connection is not fast enough to prevent two neurons from learning the same pattern. Because the TTS rule *predicts* future spikes, weight changes may occur even if, in actual fact, the output neuron does not produce an action

potential. All that is required to trigger LTP is that the membrane potential reaches the threshold imposed by the L parameter; furthermore, this prediction is based on the membrane potential prior to adding input delay corrections. Thus the TTS rule may not account for delayed inhibitory inputs, such as can happen in the WTA configuration. The result can be the simultaneous potentiation of input synapses to neurons which are rivals for the same input representation. One solution is to add more randomness in the neuron and connection parameters to break network symmetry.

Variations of the learning parameter L result in very different behaviour: while some values produce results similar to the standard learning behaviour, others have a very noisy or even an "escaping" behaviour. If we move the L parameter to a value which is too low, the network may adapt to random noise present on the input, which increases the mean membrane potential of the output neuron. In our tests no random noise was added on the input pattern. Indeed, it could modify the forecast function because of its effect upon the mean membrane potential. On the other hand, opening the forecast time window to greater values than those chosen in this chapter may lead to unpredictable and/or unstable behaviour.

The algorithm discussed in this chapter is based on a statistical approach which may be sensitive to model specifics. Statistical behaviour was extracted using only two possible types of Izhikevich neuron; changing the type of Izhikevich neuron may lead to different forecast functions. A general rule which used the neuron parameters a, b, c and d to evaluate the statistical function would be much more useful, but on the other hand there may be neurons for which no forecast function exists. Chaotic neurons may belong to this category although, for such neurons, it is questionable whether they are able to learn to respond to specific input patterns. Systematic analysis of the relationship between L, a, b, c, and d will be important future work.

Thus far, the simulations have run on the current-generation SpiNNaker board, containing 4 SpiNNaker chips with a total of 72 cores. While the simulations were executed in real-time, with the small size of network in these experiments, differences in execution speed are already significant compared to simulation on a conventional PC platform. Larger boards have already been developed having 48 chips with a total of 864 cores, and are currently under test; these boards may use dedicated board-to-board links to permit assembly into very large systems. With these boards testing of models with hundreds of thousands to millions of neurons can be tested to examine scalability systematically.

6.14 Summary

In this chapter a novel learning rule has been presented. The proposed rule has achieved a better memory occupation and a lower requirement in terms of clock cycles. However, there are some differences in the behaviour if compared with the STDP rules which have been described throughout the chapter and some improvements have been marked for future development. The next chapter summarizes the three research contributions presented, and for each of them possible developments are highlighted. In addition, a novel feature is proposed which involves the use all the three contributions for a single goal.

Chapter 7

Conclusion and future work

This chapter summarises the conclusions of the research carried out, and suggests some possible evolution. Since the topics discussed cover three different areas of research, this chapter follows the same order, summarising the conclusions and proposing possible future work for each of the research areas. In addition, one possible evolution which joins together the three topics is finally presented.

The three topics discussed involve:

1. SpikeServer: a host interface to inject spikes into the SpiNNaker system according to the SpiNNaker time clock;

2. Population-based routing: a novel multicast routing algorithm which allows very efficient computational performance and acceptable routing table space requirements.

3. The STDP TTS learning rule: a novel learning rule for spiking neural networks whose behaviour is similar to the standard STDP algorithm, but which allows faster computation and smaller memory occupation.

7.1 The SpikeServer evolution: multiple channels

The interface to inject spikes into the SpiNNaker system, the SpikeServer, was designed to provide general support to synchronize a host computer with the SpiNNaker hardware, independently of the hardware channel in use. The research presented showed that it is possible to use a non real-time channel, such as an Ethernet, to allow real-time communication with some degree of accuracy.

Currently this software is the only one developed to inject host-generated spikes in synchronism with the neural simulation on the SpiNNaker system. Therefore, for long simulations where the memory required exceeds that available by the cores, this software is a required building block.

The Ethernet interface has been used as it was the one available on the current SpiNNaker boards. However, as the development of the project proceeds, it seems reasonable that more appropriate channels should be used, so that the accuracy of the timing in the spike injection and the bandwidth of the channel improves. The SpikeServer software presented in chapter 4 has been designed modularly, so that it is possible to program new channel(s) to be used in the communication with the SpiNNaker board; in addition, in the case when multiple communication channels are required, it is possible to instantiate multiple PLL blocks, one for each instance of the channel, so that each is synchronized to a particular core. Substantially, the SpikeServer can be parallelised to multiply the input/output channels of the system, keeping a single software entry point for the user interface.

7.2 Self-configuring SpiNNaker

The population-based routing approach has been designed to connect two populations of neurons using a single channel (identified by an entry in the routing tables), leaving the detailed description of the synaptic interconnection to the very last stage, when a spike is processed by the destination core.

For routing purposes, the Longest Path First algorithm has been chosen for its performance, in terms of network resource usage, and for its capacity to create a route using only information which is local to each core.

A similar approach has been taken in the design of the software which generates the binary configuration files for the SpiNNaker system.

These specifications have been chosen so that the process of computing the routing tables, as well as all the other binary files, could be parallelised as far as possible. In fact, during the design and realization of the population-based routing principles and algorithms, as well as during the implementation of the Partition And Configuration Manager (PACMAN - see section 3.5), it was clear that the task of the binary file generation for the description of the neural networks to be simulated on SpiNNaker, is the most computationally expensive. Additionally, as highlighted on multiple occasions, the engine of the SpiNNaker simulator is a general-purpose processor which may be

	Computation requirements (mean values)		Memory requirements (see example Tab.6.2)
	Clock cycles	Time (μsec)	
Standard STDP	1,708	8.54	82,266 bytes
Spike-pair STDP	1,021	5.105	82,266 bytes
STDP TTS	270	1.35	668 bytes

Table 7.1: Comparison of computational requirements and memory occupation for the three learning algorithms available on SpiNNaker.

used for tasks different from spiking neural network simulation. The idea proposed, therefore, is to use the SpiNNaker system itself to generate such description binary files before simulation starts.

For these reasons, all algorithms implemented, so far, in the binary file generation section of PACMAN have been pipelined to reduce the memory required for each of the step of the computation. In this way it is easier to implement such algorithms on a system with a small amount of memory such as the SpiNNaker system. The key point of this process is to transfer the high-level description of the neural network to each core, and then leave the machine to self-configure. In addition, all the algorithms currently implemented using the Python scripting language need to be transferred to a language which can be compiled for ARM cores (such as the C language). This process is not trivial and requires some particular care in the management of the memory: currently this task is left to the Python interpreter, which is not available on the SpiNNaker system.

7.3 The STDP TTS learning rule

In the last chapter the STDP TTS learning rule was introduced and characterized, and the discussion showed that it is possible to use the post-synaptic membrane potential to trigger LTP. This new mechanism was shown to be computationally less expensive, keeping the learning behaviour on the selected task comparable with the standard STDP and the spike-pair STDP rules. As proof of this, the results of various tests have been presented to show, analyse and understand the behaviour of this novel learning rule, with particular regard to the newly introduced "L" parameter. In particular, Table 7.1 provides a comparison for memory occupation and computation requirements between the three learning rules available on SpiNNaker: the standard STDP rule, the spike-pair STDP rule and the STDP TTS rule.

7.4 Future work

Future work in this field is wide ranging. Some are highlighted in the following sections:

- Dynamic adaptation of the "L" parameter;
- A synaptic re-fetch event model;
- Modification of the membrane potential sampling time;

7.4.1 Dynamic adaptation of the learning parameter

As described in section 6.8, the learning parameter "L" used in the STDP TTS rule needs careful tuning to yield an algorithm which presents as little noise as possible and is able to advance in the detection of the pattern. However, from the results and the discussion presented, it is clear that the value of this parameter is connected with the mean value of the output neuron's membrane potential.

Therefore it is possible to think about a dynamic rule to compute this parameter: in the case when the mean value of the membrane potential is particularly high; the adaptation rule may increase the learning threshold so that synaptic potentiation happens more rarely. Aletrnatively, if the input synapses are less active and the mean value of the membrane potential is lower, the "L" parameter may be lowered also, so that potentiation happens more easily. This adaptation of the learning threshold may resemble (under certain aspects) the BCM (Bienenstock et al., 1982) rule, which also uses a variable threshold (θ) to trigger synaptic potentiation/depression.

Using such a rule, the neural network may adapt autonomously to a range of input synaptic weights and activity rates, even if these parameters change during the simulation.

7.4.2 The re-fetch model

The prediction of potentiation, currently, does not take into account synaptic delays, as the membrane potential is sampled at the end of the millisecond when a spike is received by the destination core. Therefore, the prediction is reliable as the synaptic delay in the current networks is $1msec$, but, as the synaptic delay increases, the forecast rule becomes less reliable.

To compensate for this problem, a software model may be implemented to keep track of all incoming spikes in a time window, and the membrane potential of each neuron for the same time window, so that is possible to defer the computation of the synaptic weight update until the membrane potential is known. However, this model would increase the core's internal memory usage and would require that the synaptic information is fetched a second time from the SDRAM when it is possible to compute the update. Therefore this model requires some performance measurement to check the applicability to a real-time system.

7.4.3 Sampling the membrane potential after the delivery of the spike

The re-fetch model may be a vehicle for other experimentation: sampling the membrane potential some milliseconds after the spike has been delivered to the post-synaptic neuron. This has been theoretically described by Harris (2008), and it may improve the learning capabilities of rules based on post-synaptic neuron membrane potential.

7.4.4 Other plasticity models

More generally, synaptic plasticity is a very wide research field, where few biological processes have been discovered through experimentation. More processes can be modelled on neuromorphic or neuromimetic hardware (such as the SpiNNaker system) to check the applicability to biology. These examples may include:

- Short-term plasticity;
- Heterosynaptic plasticity;
- Supervised learning;
- Reinforcement (reward-based) learning;
- Evolutionary learning.

7.5 Synaptic rewiring and structural plasticity

Simulators which allow structural plasticity features (also known as synaptic rewiring and synaptic pruning) are not common. In general it is plausible that software simulators are theoretically able to implement such models as they are completely programmable. Hardware simulators which do not allow on-the-fly reconfiguration of synaptic interconnections are unlikely able to simulate such feature.

However, in biology, these processes do take place on very long time scales (perhaps hours). Implementing them in a neural network simulator will allow biologists and neurologists to test models which describe how brain circuits are created, and therefore bring these models to a new scale of realism. The SpiNNaker simulator, allowing on-the-fly reconfiguration, provides a general-purpose computational substrate that is able to simulate such models.

7.5.1 Synaptogenesis

The process of generating new synapses may be simulated on SpiNNaker, taking advantage of the population-based routing principles described earlier. When a population requires a new connection with another population, during a long run of a simulation (where the input needs, therefore, to be provided through the SpikeServer), a series of events need to take place:

1. The source core, which hosts the population requiring a new connection, emits a "discovery" packet in the SpiNNaker network toward the destination population (core). The knowledge of the population mapping in the system is essential at this stage.

2. The discovery packet jumps through all the chips between the source and the destination and checks that all the links and all the routers are alive and that there is space available in the routing tables for a new connection.

3. When this packet reaches the destination core, it initialises a new memory area which will contain the information related to the new synaptic connectivity;

4. Finally, the destination core replies with an "instantiation" packet which follows the same route backwards. When each of the cores between the source and the destination receive such packet they store a new routing entry for the novel projection being created.

The projection needs to be created backwards because of the computational time required to perform all these steps: if a packet is received by the destination core, but the synaptic interconnectivity area has not yet been initialised, misbehaviours may easily happen. In the same way, if the routing path is not completely set before a packet is emitted by the source population, a system-wide misbehaviour may arise.

Once the new projection is created, synaptic plasticity may enhance or reduce the efficacy. However, given the complexity of the synaptogenic process, an inexpensive learning rule may be required to keep simulations running in real-time.

7.5.2 Synaptic pruning

This process is complementary to the one previously described. In this case the process is triggered by the destination core: a projection that has very low synaptic weights brings very low contribution to the post-synaptic population, and therefore can be pruned. The sequence of events that drives this process are:

1. The destination core sends a "request for pruning" packet to the source core. This packet passes through all intermediate cores to find the appropriate place to cut the final branch.

2. When this node is found, the routing table of the specified node is modified accordingly and a "pruning" packet is sent across the same path towards the destination core.

3. All the intermediate chips which receive such packet purge the entry in the routing table, freeing space for future new connections;

4. The pruning core also deallocates the synaptic memory area so that this memory may be allocated to other new connections.

The pruning of the multicast entry needs to be done from the router which starts the last branch of the multicast tree that reaches the core to be pruned. The pruning process then heads toward the destination, as this is the only way that packet dispersion in the system is avoided. Freeing the memory used by the synaptic interconnectivity may be done by the destination core upon the receipt of the synaptic pruning packet.

The algorithms described here are only initial thoughts on how to implement such processes. More in-depth analysis and experimental results are required to check the correctness and the possibility for optimizations.

Structural plasticity is a very complex phenomenon which starts at the beginning of a life form but is not yet completely understood. Providing a platform for simulations of this phenomenon may ease the work of neurologists and biologists who may be able to test the practicality their hypothesis and verify that they align with the reality of the events. The SpiNNaker system has been designed for such experimentations and therefore it provides a valuable help for researchers in the field.

7.6 Summary

Since the 1990s, when the synaptic plasticity process started being analysed in-depth, the rules extracted from biological observations have been the basis for experimentation in the computational neuroscience field. However, biological observations have not provided a uniquely defined rule. For example, as described in section 6.2.2, the best-known STDP rule describes synaptic efficiency variation according to precise timing of pre- and post-synaptic spike timing, but it does not define how multiple pairs are taken into account. Moreover, recent novel experimentations (Graupner and Brunel, 2012) described the mechanisms underlying this process from a different perspective, adding, in the model, calcium ions as a modulator for the process. This new model has shown to be able to incorporate all the biological results extracted until now, but still does not provide a detailed description of the mechanisms involving all the molecules at the synaptic level.

Simulators, with an increasing level of biological fidelity, have been developed using hardware and/or software techniques, comprising biologically extracted rules for in-depth analysis of the processes involved in this phenomenon. However, rules extracted directly from biological observations have always been found hard to implement in both types of simulator, therefore some of the biological details have been omitted to achieve a simpler rule which was possible to implement.

However, from the point of view of a real-time hardware/software simulator, such as SpiNNaker, the rules proposed by the biological observations are computationally very expensive. In some limit cases it may even be possible that the learning rule require more time than is available considering the real-time constraint.

The novel learning rule proposed in this research demonstrated achievable performance comparable to those defined on the basis of biological observations, but at a lower computational cost, is achievable by real-time simulators such as SpiNNaker.

Bibliography

Abarbanel, H. D. I., Huerta, R., Rabinovich, M. I., Jul. 2002. Dynamical model of long-term synaptic plasticity. Proceedings of the National Academy of Sciences 99 (15), 10132–10137.

Abbott, L. F., Nelson, S. B., Nov. 2000. Synaptic plasticity: taming the beast. Nature neuroscience 3 Suppl, 1178–1183.

Ananthanarayanan, R., Esser, S. K., Simon, H. D., Modha, D. S., 2009. The cat is out of the bag: cortical simulations with 10^9 neurons, 10^{13} synapses. In: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, New York, NY, USA.

Arena, P., Fortuna, L., Frasca, M., Patané, L., Sala, C., May 2007. Integrating high-level sensor features via STDP for bio-inspired navigation. In: Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on. IEEE, pp. 609–612.

Artola, A., Brocher, S., Singer, W., Sep. 1990. Different voltage-dependent thresholds for inducing long-term depression and long-term potentiation in slices of rat visual cortex. Nature 347 (6288), 69–72.

Bakkum, D. J., Chao, Z. C., Potter, S. M., May 2008. Long-term activity-dependent plasticity of action potential propagation delay and amplitude in cortical networks. PLoS ONE 3 (5), e2088.

Bi, G., Poo, M., 2001. Synaptic modification by correlated activity: Hebb's postulate revisited. Annual review of neuroscience 24 (1), 139–166.

Bi, G.-Q., Poo, M.-M., Dec. 1998. Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. J. Neurosci. 18 (24), 10464–10472.

Bienenstock, E. L., Cooper, L. N., Munro, P. W., Jan. 1982. Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. J. Neurosci. 2 (1), 32–48.

Binzegger, T., Douglas, R. J., Martin, K. A., Oct. 2009. Topology and dynamics of the canonical circuit of cat V1. Neural Networks 22 (8), 1071–1078.

Boahen, K. A., May 2000. Point-to-point connectivity between neuromorphic chips using address events. IEEE Trans. Circuits and Systems 2: Analog and Digital Signal Processing 47 (5), 416–434.

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J., Diesmann, M., Morrison, A., Goodman, P., Harris, F., Zirpe, M., Natschläger, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A., Boustani, S. E., Destexhe, A., Dec. 2007. Simulation of networks of spiking neurons: a review of tools and strategies. Journal of computational neuroscience 23 (3), 349–398.

Burkitt, A. N., Gilson, M., van Hemmen, J. L., May 2007. Spike-timing-dependent plasticity for neurons with recurrent connections. Biological cybernetics 96 (5), 533–546.

Cajal, S. R., Jan. 1894. The Croonian Lecture: La fine structure des centres nerveux. Proceedings of the Royal Society of London 55 (331-335), 444–468.

Carpenter, G. A., Grossberg, S., Dec. 1987. ART 2: Self-organization of stable category recognition codes for analog input patterns. Applied Optics 26 (23), 4919–4930.

Chapiro, D. M., 1984. Globally-asynchronous locally-synchronous systems. Ph.D. thesis.

Clopath, C., Busing, L., Vasilaki, E., Gerstner, W., Mar. 2010. Connectivity reflects coding: a model of voltage-based STDP with homeostasis. Nature Neuroscience 13 (3), 344–352.

Cowan, W. M., Sudhof, T. C., Stevens, C. F. (Eds.), Jan. 2001. Synapses, illustrated edition Edition. The Johns Hopkins University Press.

Cox, C. E., Blanz, W. E., Mar. 1992. GANGLION-a fast field-programmable gate array implementation of a connectionist classifier. Solid-State Circuits, IEEE Journal of 27 (3), 288–299.

Dally, W. J., Towles, B., 2004. Principles and Practices of Interconnection Networks. Morgan Kaufmann.

Darwin, C., 1859. The origin of species (6th edition). New York: P.F. Collier.

Davies, S., Galluppi, F., Rast, A. D., Furber, S. B., 2012a. A forecast-based STDP rule suitable for neuromorphic implementation. Neural Networks 32 (0), 3–14.

Davies, S., Navaridas, J., Galluppi, F., Furber, S., Jun. 2012b. Population-Based Routing in the SpiNNaker Neuromorphic Architecture. In: WCCI 2012, The 2012 IEEE World Congress on Computational Intelligence. IEEE, pp. 1932–1939.

Davies, S., Patterson, C., Galluppi, F., Rast, A. D., Lester, D., Furber, S. B., 2010. Interfacing real-time spiking I/O with the SpiNNaker neuromimetic architecture. Australian Journal of Intelligent Information Processing Systems 11 (1), 7–11.
URL `http://cs.anu.edu.au/ojs/index.php/ajiips/article/view/1071`

Davies, S., Rast, A., Galluppi, F., Furber, S., Jul. 2011a. A forecast-based biologically-plausible STDP learning rule. In: Neural Networks (IJCNN), The 2011 International Joint Conference on. IEEE, pp. 1810–1817.

Davies, S., Rast, A. D., Galluppi, F., Furber, S. B., 2011b. Maintaining real-time synchrony on SpiNNaker. In: Proceedings of the 8th ACM International Conference on Computing Frontiers. CF '11. ACM, New York, NY, USA, pp. 15:1–15:2.

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., , Yger, P., 2008. PyNN: a common interface for neuronal network simulators. Frontiers in Neuroinformatics 2.

Dayan, P., Abbott, L. F., Dec. 2001. Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems, 1st Edition. The MIT Press.

Delorme, A., 2001. Networks of integrate-and-fire neurons using Rank Order Coding B: Spike timing dependent plasticity and emergence of orientation selectivity. Neurocomputing 38-40 (1-4), 539–545.

Dimkovic, I., 2011. SpikeFun [online] (accessed on July 1st, 2012).
URL http://www.dimkovic.com/

Draghici, S., Feb. 2000. Neural networks in analog hardware–design and implementation issues. International journal of neural systems 10 (1), 19–42.

Eliasmith, C., Anderson, C. H., Aug. 2004. Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems (Computational Neuroscience), new ed Edition. A Bradford Book.

Farries, M. A., Fairhall, A. L., Dec. 2007. Reinforcement Learning With Modulated Spike TimingDependent Synaptic Plasticity. Journal of Neurophysiology 98 (6), 3648–3665.

Fingelkurts, A. A., Fingelkurts, A. A., Kähkönen, S., Jan. 2005. Functional connectivity in the brain–is it an elusive concept? Neuroscience and biobehavioral reviews 28 (8), 827–836.

Furber, S., Nov. 2008. The future of computer technology and its implications for the computer industry. The Computer Journal 51 (6), 735–740.

Furber, S., Jan. 2011. SpiNNaker Data Sheet v2.02.

Furber, S. B., Lester, D. R., Plana, L. A., Garside, J. D., Painkras, E., Temple, S., Brown, A. D., 2012. Overview of the SpiNNaker System Architecture. IEEE Transactions on Computers (PrePrints).

Furber, S. B., Temple, S., Brown, A. D., Apr. 2006. High-performance computing for systems of spiking neurons. In: Proc. AISB'06 workshop on GC5: Architecture of Brain and Mind.

Galluppi, F., Davies, S., Rast, A. D., Sharp, T., Plana, L., Furber, S., 2012a. A Hierachical Configuration System for a Massively Parallel Neural Hardware Platform. In: Proceedings of the 9th ACM international conference on Computing frontiers. ACM.

Galluppi, F., Davies, S., Stewart, T., Eliasmith, C., Furber, S., Jun. 2012b. Real Time On-Chip Implementation of Dynamical Systems with Spiking Neurons. In: WCCI 2012, The 2012 IEEE World Congress on Computational Intelligence. IEEE, pp. 2455–2462.

Galluppi, F., Rast, A., Davies, S., Furber, S., 2010. A general-purpose model translation system for a universal neural chip. In: Proceedings of the 17th international conference on Neural information processing: theory and algorithms - Volume Part I. ICONIP'10. Springer-Verlag, Berlin, Heidelberg, pp. 58–65.

Garm, A., O'Connor, M., Parkefelt, L., Nilsson, D. E., Oct. 2007. Visually guided obstacle avoidance in the box jellyfish tripedalia cystophora and chiropsella bronzie. Journal of Experimental Biology 210 (20), 3616–3623.

Gerstner, W., Kistler, W. M., Dec. 2002. Mathematical formulations of Hebbian learning. Biological Cybernetics 87 (5-6), 404–415.

Gilson, M., Burkitt, A. N., Grayden, D. B., Thomas, D. A., van Hemmen, J. L., Dec. 2009. Emergence of network structure due to spike-timing-dependent plasticity in recurrent neuronal networks IV. Biological Cybernetics 101 (5-6), 427–444.

Goodman, D., Brette, R., 2008. Brian: a simulator for spiking neural networks in python. Frontiers in neuroinformatics 2.

Graupner, M., Brunel, N., Feb. 2012. Calcium-based plasticity model explains sensitivity of synaptic changes to spike pattern, rate, and dendritic location. Proceedings of the National Academy of Sciences 109 (10), 3991–3996.

Grossberg, S., Versace, M., Jul. 2008. Spikes, synchrony, and attentive learning by laminar thalamocortical circuits. Brain Research 1218, 278–312.

Guyonneau, R., VanRullen, R., Thorpe, S. J., Apr. 2005. Neurons Tune to the Earliest Spikes Through STDP. Neural Computation 17 (4), 859–879.

Hagmann, P., Cammoun, L., Gigandet, X., Meuli, R., Honey, C. J., Wedeen, V. J., Sporns, O., 2008. Mapping the structural core of human cerebral cortex. PLoS Biol 6 (7).

Harris, K. D., Mar. 2008. Stability of the fittest: organizing learning through retroaxonal signals. Trends Neurosci 31 (3), 130–136.

Haykin, S. S., Jul. 1999. Neural networks: a comprehensive foundation, 2nd Edition. Prentice Hall, Ch. 2.

Hebb, D. O., 1949. The Organization of Behavior: A Neuropsychological Theory. Wiley.

Holland, J., 1975. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI, USA.

Humphries, M. D., Gurney, K., Oct. 2007. Solution Methods for a New Class of Simple Model Neurons. Neural Computation 19 (12), 3216–3225.

Indiveri, G., Chicca, E., Douglas, R., Jan. 2006. A VLSI Array of Low-Power Spiking Neurons and Bistable Synapses With Spike-Timing Dependent Plasticity. IEEE Transactions on Neural Networks 17 (1), 211–221.

Indiveri, G., Chicca, E., Douglas, R. J., Jun. 2009. Artificial Cognitive Systems: From VLSI Networks of Spiking Neurons to Neuromorphic Cognition. Cognitive Computation 1 (2), 119–127.

Izhikevich, E. M., 2003. Simple model of spiking neurons. IEEE Trans. Neural Networks, 1569–1572.

Izhikevich, E. M., Sep. 2004. Which model to use for cortical spiking neurons? IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council 15 (5), 1063–1070.

Izhikevich, E. M., Feb. 2006. Polychronization: Computation with Spikes. Neural Computation 18 (2), 245–282.

Izhikevich, E. M., Nov. 2010. Hybrid spiking models. Philosophical transactions. Series A, Mathematical, physical, and engineering sciences 368 (1930), 5061–5070.

Izhikevich, E. M., Desai, N. S., Jul. 2003. Relating STDP to BCM. Neural Computation 15 (7), 1511–1523.

James, M., Hoang, D., Mar. 1992. Design of low-cost, real-time simulation systems for large neural networks. J. Parallel and Distributed Computing 14 (3), 221–235.

Jin, X., Furber, S. B., Woods, J. V., Jun. 2008. Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In: Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on. pp. 2812–2819.

Jin, X., Galluppi, F., Patterson, C., Rast, A., Davies, S., Temple, S., Furber, S., Jul. 2010a. Algorithm and software for simulation of spiking neural networks on the

multi-chip SpiNNaker system. In: Neural Networks (IJCNN), The 2010 International Joint Conference on. IEEE, pp. 649–656.

Jin, X., Lujan, M., Plana, L. A., Davies, S., Temple, S., Furber, S. B., 2010b. Modeling spiking neural networks on SpiNNaker. Computing in Science & Engineering 12 (5), 91–97.

Jin, X., Rast, A., Galluppi, F., Davies, S., Furber, S., Jul. 2010c. Implementing spike-timing-dependent plasticity on SpiNNaker neuromorphic hardware. In: Neural Networks (IJCNN), The 2010 International Joint Conference on. IEEE, pp. 2302–2309.

Jin, X., Rast, A., Galluppi, F., Khan, M., Furber, S., 2009. Implementing learning on the SpiNNaker universal neural chip multiprocessor. In: Leung, C. S., Lee, M., Chan, J. H. (Eds.), Neural Information Processing. Vol. 5863. Springer, Berlin, Heidelberg, Ch. 48, pp. 425–432.

Kandel, E. R., Schwartz, J. H., Jessell, T. M., Jul. 2000. Principles of Neural Science, 4th Edition. McGraw-Hill Medical, Ch. 17.

Kube, K., Herzog, A., Michaelis, B., de Lima, A. D., Voigt, T., Jul. 2008. Spike-timing-dependent plasticity in small-world networks. Neurocomputing 71 (7), 1694–1704.

Lefort, S., Tomm, C., Floyd Sarria, J.-C., Petersen, C. C. H., Jan. 2009. The excitatory neuronal network of the C2 barrel column in mouse primary somatosensory cortex. Neuron 61 (2), 301–316.

Leñero-Bardallo, J. A., Serrano-Gotarredona, T., Linares-Barranco, B., Oct. 2010. A Five-Decade Dynamic-Range Ambient-Light-Independent Calibrated Signed-Spatial-Contrast AER Retina With 0.1-ms Latency and Optional Time-to-First-Spike Mode. IEEE Transactions on Circuits and Systems I: Regular Papers 57 (10), 2632–2643.

Lichtsteiner, P., Posch, C., Delbruck, T., 2008. A 128x128 120dB 15us Latency Asynchronous Temporal Contrast Vision Sensor. IEEE J. Solid State Circuits 43, 566–576.

Lindsey, C. S., Lindblad, T., 1995. Survey of neural network hardware. In: Proc. SPIE, Applications and Science of Artificial Neural Networks. Vol. 2492. pp. 1194–1205.

Liu, J. W. S., Apr. 2000. Real-Time Systems, 1st Edition. Prentice Hall.

Livi, P., Indiveri, G., May 2009. A current-mode conductance-based silicon neuron for address-event neuromorphic systems. In: 2009 IEEE International Symposium on Circuits and Systems. IEEE, pp. 2898–2901.

Lysecky, R., Vahid, F., 2003. On-chip Logic Minimization. In: Proceedings of the 40 th Design Automation Conference (DAC. pp. 334–337.

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

MacNeil, D., Eliasmith, C., Sep. 2011. Fine-Tuning and the Stability of Recurrent Neural Networks. PLoS ONE 6 (9), e22885.

Maguire, L., McGinnity, T. M., Glackin, B., Ghani, A., Belatreche, A., Harkin, J., Dec. 2007. Challenges for large-scale implementations of spiking neural networks on FPGAs. Neurocomputing 71 (1-3), 13–29.

Malsburg, C., Jun. 1973. Self-organization of orientation sensitive cells in the striate cortex. Biological Cybernetics 14 (2), 85–100.

Markram, H., Feb. 2006. The blue brain project. Nat Rev Neurosci 7 (2), 153–60.

Markram, H., Gerstner, W., Sjöström, P. J., 2012. Spike-Timing-Dependent Plasticity: A Comprehensive Overview. Frontiers in synaptic neuroscience 4.

Markram, H., Tsodyks, M., Aug. 1996. Redistribution of synaptic efficacy between neocortical pyramidal neurons. Nature 382 (6594), 807–810.

Masquelier, T., Guyonneau, R., Thorpe, S. J., Dec. 2008a. Competitive STDP-Based Spike Pattern Learning. Neural Computation 21 (5), 1259–1276.

Masquelier, T., Guyonneau, R., Thorpe, S. J., Jan. 2008b. Spike Timing Dependent Plasticity Finds the Start of Repeating Patterns in Continuous Spike Trains. PLoS ONE 3 (1), e1377.

Masquelier, T., Thorpe, S. J., 02 2007. Unsupervised Learning of Visual Features through Spike Timing Dependent Plasticity. PLoS Comput Biol 3 (2), e31.

Masuda, N., Kori, H., Jun. 2007. Formation of feedforward networks and frequency synchrony by spike-timing-dependent plasticity. Journal of Computational Neuroscience 22 (3), 327–345.

Merolla, P., Arthur, J., Akopyan, F., Imam, N., Manohar, R., Modha, D. S., Sep. 2011. A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm. In: 2011 IEEE Custom Integrated Circuits Conference (CICC). IEEE, pp. 1–4.

Merolla, P. A., Arthur, J. V., Shi, B. E., Boahen, K. A., Feb. 2007. Expandable Networks for Neuromorphic Chips. Circuits and Systems I: Regular Papers, IEEE Transactions on 54 (2), 301–311.

Misra, J., Saha, I., Dec. 2010. Artificial neural networks in hardware: A survey of two decades of progress. Neurocomputing 74 (1-3), 239–255.

Nolfi, S., Parisi, D., Elman, J. L., Jun. 1994. Learning and evolution in neural networks. Adaptive Behavior 3 (1), 5–28.

Pakkenberg, B., Pelvig, D., Marner, L., Bundgaard, M. J., Gundersen, H. J. J., Nyengaard, J. R., Regeur, L., 2003. Aging and the human neocortex. Experimental gerontology 38 (1-2), 95–99.

Pavlidis, N. G., Tasoulis, O. K., Plagianakos, V. P., Nikiforidis, G., Vrahatis, M. N., Jul. 2005. Spiking neural network training using evolutionary algorithms. In: Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on. Vol. 4. IEEE, pp. 2190–2194 vol. 4.

Pfister, J.-P. P., Gerstner, W., Sep. 2006. Triplets of spikes in a model of spike timing-dependent plasticity. The Journal of Neuroscience : the official journal of the Society for Neuroscience 26 (38), 9673–9682.

Plana, L. A., Furber, S. B., Temple, S., Khan, M., Shi, Y., Wu, J., Yang, S., Oct. 2007. A GALS Infrastructure for a Massively Parallel Multiprocessor. Design & Test of Computers, IEEE 24 (5), 454–463.

Ponulak, F., Kasinski, A., 2011. Introduction to spiking neural networks: Information processing, learning and applications. Acta neurobiologiae experimentalis 71 (4), 409–433.

Rao, R. P. N., Sejnowski, T. J., Oct. 2001. Spike-Timing-Dependent Hebbian Plasticity as Temporal Difference Learning. Neural Computation 13 (10), 2221–2237.

Rast, A., 2010. Scalable event-driven modelling architectures for neuromimetic hardware. Ph.D. thesis.

Rast, A., Galluppi, F., Davies, S., Plana, L., Patterson, C., Sharp, T., Lester, D., Furber, S., Nov. 2011a. Concurrent heterogeneous neural model simulation on real-time neuromimetic hardware. Neural Networks 24 (9), 961–978.

Rast, A., Galluppi, F., Davies, S., Plana, L. A., Sharp, T., Furber, S., Jul. 2011b. An event-driven model for the SpiNNaker virtual synaptic channel. In: Neural Networks (IJCNN), The 2011 International Joint Conference on. IEEE, pp. 1967–1974.

Rast, A., Jin, X., Khan, M., Furber, S., 2009. The deferred-event model for hardware-oriented spiking neural networks. In: Proc. 2008 Int'l Conf. Neural Information Processing (ICONIP 2008). Springer-Verlag, pp. 1057–1064.

Rast, A. D., Galluppi, F., Jin, X., Furber, S., 2010a. The Leaky Integrate-and-Fire Neuron: A Platform for Synaptic Model Exploration on the SpiNNaker Chip. Neural Networks, 2010. IJCNN 2010. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on.

Rast, A. D., Galluppi, F., Jin, X., Furber, S. B., Jul. 2010b. The Leaky Integrate-and-Fire neuron: A platform for synaptic model exploration on the SpiNNaker chip. In: The 2010 International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1–8.

Rast, A. D., Jin, X., Galluppi, F., Plana, L. A., Patterson, C., Furber, S., 2010c. Scalable event-driven native parallel processing: the SpiNNaker neuromimetic system. In: Proceedings of the 7th ACM international conference on Computing frontiers. CF '10. ACM, New York, NY, USA, pp. 21–30.

Roberts, P. D., Bell, C. C., Dec. 2002. Spike timing dependent synaptic plasticity in biological systems. Biological cybernetics 87 (5-6), 392–403.

Sanes, D. H., Reh, T. A., Harris, W. A., 2000. Development of the nervous system. Academic press, Ch. 1.

Schemmel, J., Bruderle, D., Grubl, A., Hock, M., Meier, K., Millner, S., May 2010. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In: Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on. IEEE, pp. 1947–1950.

Sejnowski, T. J., Nov. 1977. Statistical constraints on synaptic plasticity. Journal of theoretical biology 69 (2), 385–389.

Seo, J., Brezzo, B., Liu, Y., Parker, B. D., Esser, S. K., Montoye, R. K., Rajendran, B., Tierno, J. A., Chang, L., Modha, D. S., Friedman, D. J., Sep. 2011. A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In: 2011 IEEE Custom Integrated Circuits Conference (CICC). IEEE, pp. 1–4.

Silver, R., Boahen, K., Grillner, S., Kopell, N., Olsen, K. L., Oct. 2007. Neurotech for Neuroscience: Unifying Concepts, Organizing Principles, and Emerging Tools. The Journal of Neuroscience 27 (44), 11807–11819.

Sjöström, J., Gerstner, W., 2010. Spike-timing dependent plasticity. Scholarpedia 5 (2), 1362.

Song, S., Abbott, L. F., Oct. 2001. Cortical development and remapping through spike timing-dependent plasticity. Neuron 32 (2), 339–350.

Song, S., Miller, K. D., Abbott, L. F., Sep. 2000. Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. Nature neuroscience 3 (9), 919–926.

Stork, D. G., 1989. Is backpropagation biologically plausible? In: Neural Networks, 1989. IJCNN., International Joint Conference on. Vol. 2. IEEE, pp. 241–246.

Strain, T. J., McDaid, L. J., Maguire, L. P., McGinnity, T. M., 2006. A Supervised STDP Based Training Algorithm with Dynamic Threshold Neurons. In: Neural Networks, 2006. IJCNN '06. International Joint Conference on. IEEE, pp. 3409–3414.

Thomson, A. M., Lamy, C., Nov. 2007. Functional maps of neocortical local circuitry. Frontiers in neuroscience 1 (1), 19–42.

Thorpe, S., Gautrais, J., 1998. Rank order coding. In: Proceedings of the sixth annual conference on Computational neuroscience : trends in research, 1998: trends in research, 1998. CNS '97. Plenum Press, New York, NY, USA, pp. 113–118.

Turrigiano, G. G., May 1999. Homeostatic plasticity in neuronal networks: the more things change, the more they stay the same. Trends in Neurosciences 22 (5), 221–227.

van Ooyena, A., Roelfsemab, P. R., Jun. 2003. A biologically plausible implementation of error-backpropagation for classification tasks. In: Kaynak, O., Alpaydin, E., Oja, E., Xu, L. (Eds.), Artificial Neural Networks and Neural Information Processing - Supplementary Proceedings ICANN/ICONIP 2003. pp. 442–444.

van Schaik, A., Liu, S.-C., 2005. AER EAR: A matched silicon cochlea pair with address event representation interface. In: IEEE International Symposium on Circuits And Systems. pp. 4213–4216.

Wang, H.-X. X., Gerkin, R. C., Nauen, D. W., Bi, G.-Q. Q., Feb. 2005. Coactivation and timing-dependent integration of synaptic potentiation and depression. Nature neuroscience 8 (2), 187–193.

Watkins, C., Dayan, P., May 1992. Technical Note: Q-Learning. Machine Learning 8 (3), 279–292.

Webb, A., Davies, S., Lester, D., 2011. Spiking neural PID controllers neural information processing. In: Lu, B.-L., Zhang, L., Kwok, J. (Eds.), Neural Information Processing. Vol. 7064 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Berlin, Heidelberg, pp. 259–267.

Williams, R. W., Herrup, K., 1988. The control of neuron number. Annual review of neuroscience 11 (1), 423–453.

Willshaw, D., Dayan, P., Mar. 1990. Optimal plasticity from matrix memories: What goes up must come down. Neural Comput. 2, 85–93.

Wilson, M. A., Bhalla, U. S., Uhley, J. D., Bower, J. M., 1989. Advances in neural information processing systems 1. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Ch. GENESIS: a system for simulating neural networks, pp. 485–492.

Wu, J., Furber, S., Mar. 2010. A Multicast Routing Scheme for a Universal Spiking Neural Network Architecture. Comput. J. 53 (3), 280–288.

Wu, J., Furber, S., Garside, J., 2009. A Programmable Adaptive Router for a GALS Parallel System. Asynchronous Circuits and Systems, International Symposium on 0, 23–31.

Zhang, M., Fulcher, J., May 1996. Face recognition using artificial neural network group-based adaptive tolerance (GAT) trees. Neural Networks, IEEE Transactions on 7 (3), 555–567.

Zhu, J., Sutton, P., 2003. FPGA Implementations of Neural Networks - A Survey of a Decade of Progress. In: Field-Programmable Logic and Applications. pp. 1062–1066.

Zucker, R. S., Regehr, W. G., 2002. Short-term synaptic plasticity. Annual review of physiology 64 (1), 355–405.