

A hardware implementation of a network of functional spiking neurons with hebbian learning

Andrés Upegui, Carlos Andrés Peña-Reyes, Eduardo Sánchez

Swiss Federal Institute of Technology, Logic Systems Laboratory,
1015 Lausanne, Switzerland
(andres.uegui, carlos.pena, eduardo.sanchez)@epfl.ch

Abstract. In this paper we present a functional model of a spiking neuron intended for hardware implementation. Some features of biological spiking neurons are abstracted, while preserving the functionality of the network, in order to define an architecture with low implementation cost in field programmable gate arrays (FPGAs). Adaptation of synaptic weights is implemented with hebbian learning. As an example application we present a frequency discriminator to verify the computing capabilities of a generic network of our neuron model.

1 Introduction

The human brain contains more than 10^{11} neurons connected among them in an intricate network. Communication between neurons is done by spikes, where millions are emitted each second in every volume of cortex. Issues like the information contained in such a spatio-temporal pattern of pulses, the code used by the neurons to transmit information, or the decoding of the signal by receptive neurons, have a fundamental importance in the problem of neuronal coding. They are, however, still not fully resolved. [1]

Biological neurons are extremely complex biophysical and biochemical entities. Before designing a model it is therefore necessary to develop an intuition for what is important and what can be safely neglected. Biological models attempt to describe the neuron response, as truthful as possible, in terms of biologic parameters, such as ion channels conductances or dendrites lengths. However, biological models are not suitable for computational purposes, because of that phenomenological neural models are proposed extracting the most relevant features from their biological counterparts.

As for any model of neuron, adaptivity is required for a network of these neural models. Understanding adaptivity as any modification performed on a network to perform a given task, several types of methods could be identified, according to the type of modification done. The most common methods modify either the synaptic weights [2] or/and the network topology [5,11].

Synaptic weight modification is the most widely used approach, as it provides a relatively smooth search space. On the other hand, the sole topology modification produces a search space with a highly rugged landscape (i.e. a small change on the network results on very different performances), and although this type of adaptation allows to explore well the space of computational capabilities of the network, it is

difficult to find a solution. Growing [5], pruning, and genetic algorithms [11] are adaptive methods that modify a network topology.

A hybrid of both methods could achieve better performances, because the weight-adaptation method contributes to smooth the search space rendering easier to find a solution. We propose, thus, a hybrid method where an adaptation of the structure is done by modifying the network topology, allowing the exploration of different computational capabilities. The evaluation of these capabilities is done by weight-learning, finding in this way a solution for the problem. However, topology modification implies a high computational cost. Besides the fact that weight learning can be time-consuming, it would be multiplied by the number of topologies that are going to be explored. Under these conditions, on-line applications would be unfeasible, unless it is available enough knowledge of the problem in order to restrict the search space just to tune certain small modifications on the topology.

A part of the problem can be solved with a hardware implementation: in this case the execution time is highly reduced since the evaluation of the network is performed with the neurons running in parallel. A complexity problem remains: while on software, extra neurons and connections imply just some extra loops, in hardware implementation there is a limited area that bounds the number of neurons that can be placed on a network. This is due to the fact that each neuron has a physical existence, occupying a given area and that each connection implies a physical cable that must connect two neurons. Moreover, if an exploration of topologies is done, the physical resources (connections and neurons) for the most complex possible networks must be allocated in advance, even if the final solution is less complex. This fact renders the connectionism a critical issue since a connection matrix for a high amount of neurons is considerably resource-consuming.

Recent FPGAs allow tackling this resource availability problem thanks to their dynamic partial reconfiguration (DPR) feature, which allows the reusing of internal logic resources. This feature permits to dynamically reconfigure the same physical logic units with different configurations, reducing the size of the hardware requirements, and optimizing the number of neurons and the connectivity resources.

Topology evolution with DPR is part of our project but it is not presented in this paper, a description of it can be found in [9]. The DPR feature allows having a modular system as that described in Figure 1: different possible configurations are available for each module, being possible to communicate with neighbor modules, allowing spikes to be transmitted forward or backward for recurrent networks. Modules contain layers of neurons with a predefined connectionism, and a genetic algorithm would search for the best combination of layers.

In this paper we present a hardware-architecture for a functional neuron with hebbian learning. A neural network is implemented on an FPGA to check the amount of neurons with a given connectionism that could be contained on a given device. The network topology corresponds to the module-based system that we have just described. In order to verify the learning capabilities of our network, a dynamic problem is assigned to the network: discrimination of two signals with different frequencies.

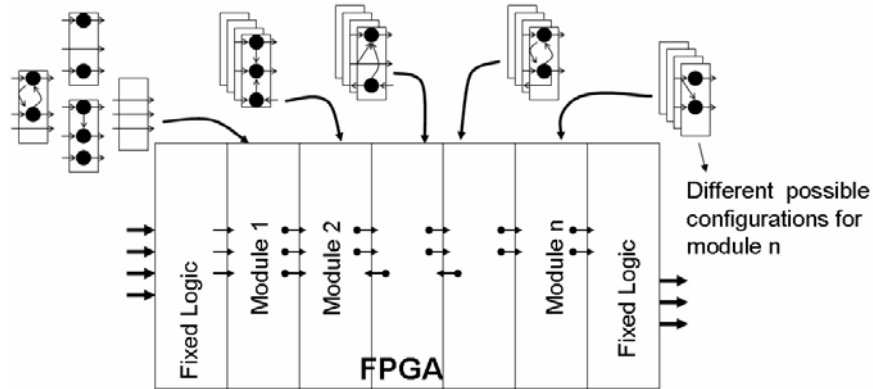


Figure 1. Layout of our reconfigurable network topology. Fixed and reconfigurable modules are allowed. In this example, fixed modules code and decode inputs and outputs to spikes, while reconfigurable modules contain network layers.

2. Neural Models

Most neuron models, such as perceptron or radial basis functions, use continuous values as inputs and outputs, processed using logistic, gaussian or other continuous functions [5,2]. In contrast, biological neurons process pulses: as a neuron receives input pulses by its dendrites, its membrane potential increases according to a post-synaptic response. When the membrane potential reaches a certain threshold value, the neuron fires, and generates an output pulse through the axon. The best known biological model is the Hodgkin and Huxley model (H&H) [3], which is based on ion current activities through the neuron membrane.

However, the most biologically plausible models are not well suited for computational implementations. This is the reason why other different approaches are needed. The *leaky integrate and fire* (LI&F) model [1,4] is based on a current integrator, modeled as a resistance and a capacitor in parallel. Differential equations describe the voltage given by the capacitor charge, and when a certain voltage is reached the neuron fires. The *spike response model order 0* (SRM_0) [1,4] offers a resembling response to that of the LI&F model, with the difference that, in this case, the membrane potential is expressed in terms of kernel functions instead of differential equations.

Spiking-neuron models process discrete values representing the presence or absence of spikes; this fact allows a simple connectionism structure at the network level and a striking simplicity at the neuron level. However, implementing models like SRM_0 and LI&F on digital hardware is largely inefficient, wasting many hardware resources and exhibiting a large latency due to the implementation of kernels and numeric integrations. This is why a functional hardware-oriented model is necessary to achieve fast architectures at a reasonable chip area cost.

2.1. The proposed neuron model

Our simplified integrate and fire model [8], as standard spiking models, uses the following five concepts: (1) membrane potential, (2) resting potential, (3) threshold potential, (4) postsynaptic response, and (5) after-spike response (see figure 2). A spike is represented by a pulse. The model is implemented as a Moore finite state machine. Two states, operational and refractory, are allowed.

During the operational state, the membrane potential is increased (or decreased) each time a pulse is received by an excitatory (or inhibitory) synapse, and then it decreases (or increases) with a constant slope until the arrival to the resting value. If a pulse arrives when a previous postsynaptic potential is still active, its action is added to the previous one. When a firing condition is fulfilled (i.e., potential \geq threshold) the neuron fires, the potential takes on a hyperpolarization value called after-spike potential and the neuron passes then to the refractory state.

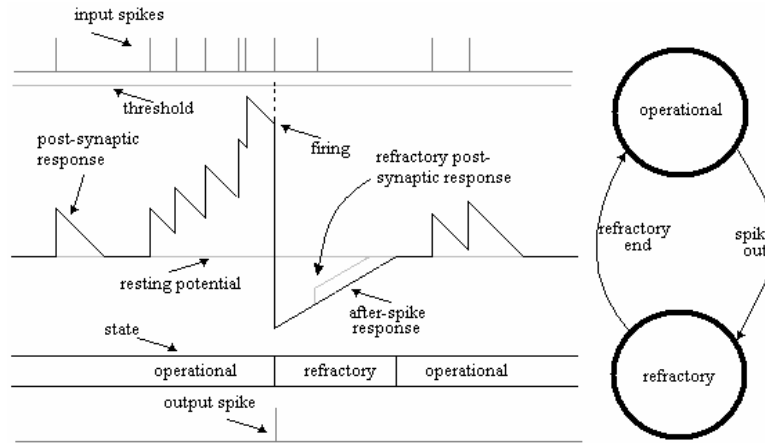


Figure 2. Response of the model to a train of input spikes.

After firing, the neuron enters a refractory period in which it recovers from the after-spike potential to the resting potential. Two kinds of refractoriness are allowed: absolute and partial. Under absolute refractoriness, input spikes are ignored. Under partial refractoriness, the effect of input spikes is attenuated by a constant factor. The refractory state acts like a timer that determines the time needed by a neuron to recover from a firing; the time is completed when the membrane potential reaches the resting potential, and the neuron comes back to the operational state.

Our model simplifies some features with respect to SRM_0 and LI&F, mainly, the post-synaptic response. The way in which several input spikes are processed affects the dynamics of the system: under the presence of 2 simultaneous input spikes, SRM_0 performs a linear superposition of post-synaptic responses, while our model adds the synaptic weights to the membrane potential. Even though our model is less biologically plausible than SRM_0 and LI&F, it is still functionally adequate.

2.2 Learning

Weight learning is an issue that has not been satisfactorily solved on spiking neuron models. Several learning rules had been explored by researchers, being hebbian learning one of the most studied [1,4]. Hebbian learning modifies the synaptic weight W_{ij} , considering the simultaneity of the firing times of the pre- and post-synaptic neurons i and j . Herein we will describe an implementation of hebbian learning oriented to digital hardware. Two modules are added to the neuron presented above: the *active-window* module and the *learning* module.

The *active-window* module determines if a given neuron learning-window is active or not (Figure 3). The module activates the output *active window* (aw_i) during a certain time after the generation of a spike by a neuron n_i . The aw_i signal is given by $aw_i = \text{step}(t_i) - \text{step}(t_i + w)$, where t_i is the firing time of n_i and w is the size of the learning window. This window allows the receptor neuron (n_j) to determine the synaptic weight modification (ΔW_{ij}) that must be done.

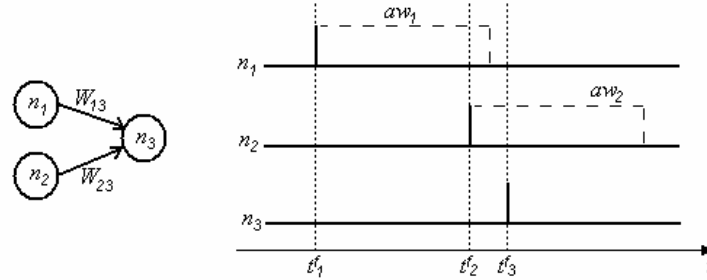


Figure 3. Hebbian learning windows. When neuron 3 fires at t_3 , the learning window of neurons n_1 and n_2 are disabled and enabled respectively. At time t_3 Synaptic weight W_{13} is decreased by the learning algorithm, while W_{23} is increased.

The *learning* module modifies the synaptic weights of the neuron, performing the hebbian learning (Figure 3). Given a neuron (n_i) with k inputs, when a firing is performed by n_i , the learning modifies the synaptic weights (W_{ij} with $j=1,2,\dots,k$). We must define in advance a learning rate α and a decay rate β , to obtain the expression: $\Delta W_{ij} = \alpha aw_j - \beta$, where α and β are positive constants and $\alpha > \beta$.

These two modules, *active-window* and *learning*, increase the amount of interneuron connectivity as they imply, respectively, one extra output and k extra inputs for a neuron.

3. The proposed neuron on hardware

Several hardware implementations of spiking neurons have been developed on analog and digital circuits. Analog electronic neurons achieve postsynaptic responses very similar to their biological counterpart; however, analog circuits use to be difficult to setup and debug. On the other hand, digital spiking neurons, use to be less biologically plausible, but are easier to setup, debug, scale, and learn, among other features.

Additionally these models can be rapidly prototyped and tested thanks to configurable logic devices such as field programmable gate arrays (FPGAs).

3.1 Field programmable gate arrays

An FPGA circuit is an array of logic cells placed in an infrastructure of interconnections. Each logic cell is a universal function or a functionally complete logic device that can be programmed by a configuration bitstream, as well as interconnections between cells, to realize a certain function [7]. Some FPGAs allow performing dynamic partial reconfiguration (DPR), where a reduced bitstream reconfigures just a given subset of internal components. DPR is done with the device active: certain areas of the device can be reconfigured while other areas remain operational and unaffected by the reprogramming.

3.2 Implementation of our model

The hardware implementation of our neuron model is illustrated in Figure 4. The computing of a time slice (iteration) is given by a pulse at the input `clk_div`, and takes a certain number of clock cycles depending on the number of inputs to the neuron. The synaptic weights are stored on a memory, which is swept by a counter. Under the presence of an input spike, its respective weight is enabled to be added to the membrane potential. Likely, the decreasing and increasing slopes (for the post-synaptic and after-spike responses respectively) are contained in the memory.

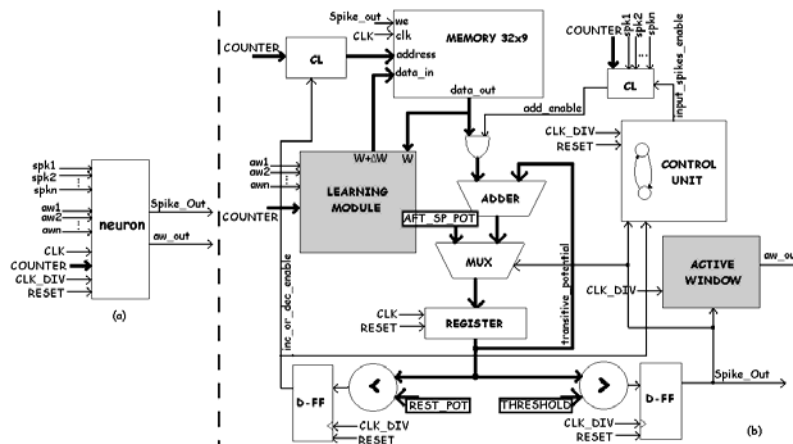


Figure 4. (a) External view of the neuron. (b) Architecture of a neuron.

Although the number of inputs to the neuron is parameterizable, increasing the number of inputs implies raising both: the area cost and the latency of the system. Indeed, the area cost highly depends on the memory size, which itself depends on the number of inputs to the neuron (e.g. the 32x9-neuron on figure 4 has a memory size

of 32x9 bits, where the 32 positions correspond to up to 30 input weights and the increasing and decreasing slopes; 9 bits is the arbitrarily chosen data-bus size). The time required to compute a time slice is equivalent to the number of inputs +1 –i.e. 30 inputs plus either the increasing or the decreasing slope. (More details on the neuron architecture can be found on [9].)

The dark blocks on Figure 4, *active-window* and *learning module*, perform the learning on the neuron. The *active window* block consists on a counter that starts a count when an output spike is generated, and stops when a certain value is reached. This value constitutes the learning window. The output *aw_out* values logic-0 if a stop condition is met for the counter and logic-1 otherwise.

The *learning* module is the one who performs the synaptic weight learning. This module computes the change to be applied to the weights (ΔW), provided that the respective learning window is active, and maintaining the weights bounded. At each clock cycle the module computes the new weight for the synapse pointed by the *COUNTER* signal; however, these new weights are stored only when an output spike is generated by the current neuron, which enables the *write-enable* on the memory.

4. Experimental setup and results

The experimental setup consists of two parts: the synthesis of a spiking neural network on an FPGA and a simulation of this network solving a problem of frequency discrimination running on Matlab.

4.1 The network on hardware

A neural network was implemented on an FPGA in order to check the number of neurons that we could include on a network. We worked with a Spartan II xc2s200 FPGA from Xilinx Corp.[10] with a maximum capacity of implementing until 200.000 logic gates. This FPGA has a matrix of 28 x 42 configurable logic blocks CLBs, each one of them containing 2 slices, which contain the logic where the functions are implemented, for a total of 2352 slices. The xc2s200 is the largest device from the low cost FPGA family Spartan II. Other FPGAs families such as Virtex II offer up to 40 times more logic resources.

We implemented the 30-input neuron described in Figure 4 both: with and without learning (Table 1). Without learning the neuron used 23 slices (0.98% of the whole FPGA), while with the learning modules it used 41 slices (1.74%). In order to observe the scalability of the neuron model, we implemented also two other non-learning neurons. A 14-input neuron (requiring a smaller memory to store the weights) needed 17 slices (0.72%), and a 62-input neuron used 46 slices (1.95%). The resources required by our neurons are very low as compared to other more biologically-plausible implementations (e.g., Ros et al [6] use 7331 Virtex-E CLB slices for 2 neurons).

Table1. Synthesis results for the neurons.

| Non-learning neurons | | | Learning neurons | |
|----------------------|-----------------|-----------------|------------------|------------------------------------|
| 14-input neuron | 30-input neuron | 62-input neuron | 30-input neuron | Full network (30 30-input neurons) |
| 17 slices | 23 slices | 46 slices | 41 slices | 1350 slices |
| 0.72% | 0.98% | 1.95% | 1.74% | 57.4% |

Using the 30-input neuron, we implemented a network with 3 layers, each layer containing 10 neurons that are internally full-connected. Additionally, layers provide outputs to the preceding and the following layers, and receive outputs from them, as described in Figure 5. For the sake of modularity, each neuron has 30 inputs: 10 from its own layer, 10 from the preceding one, and 10 from the next one. When implementing, a single layer uses 450 slices (19.13%), and the full network, with 30 neurons, needs 1350 slices (57.4%).

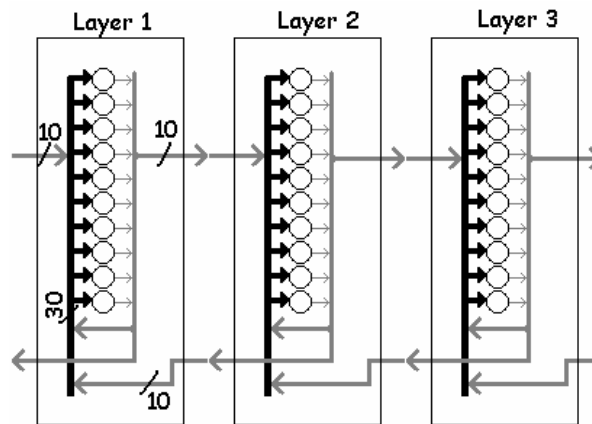


Figure 5. Layout of the network implemented on hardware.

4.1 Application: A frequency discriminator

A frequency discriminator was implemented in order to test the capability of the learning network to unsupervisedly solve a problem with dynamical characteristics. We used the network described in the previous subsection, with the following considerations for data presentation: (1) we use 9 inputs at layer 1 to introduce the pattern, (2) two sinusoidal waveforms with different periods are generated, (3) the waveforms are normalized and discretized to 9 levels, (4) a spike is generated every 3 time slices (iterations) at the input that address the discretized signal (Figure 6).

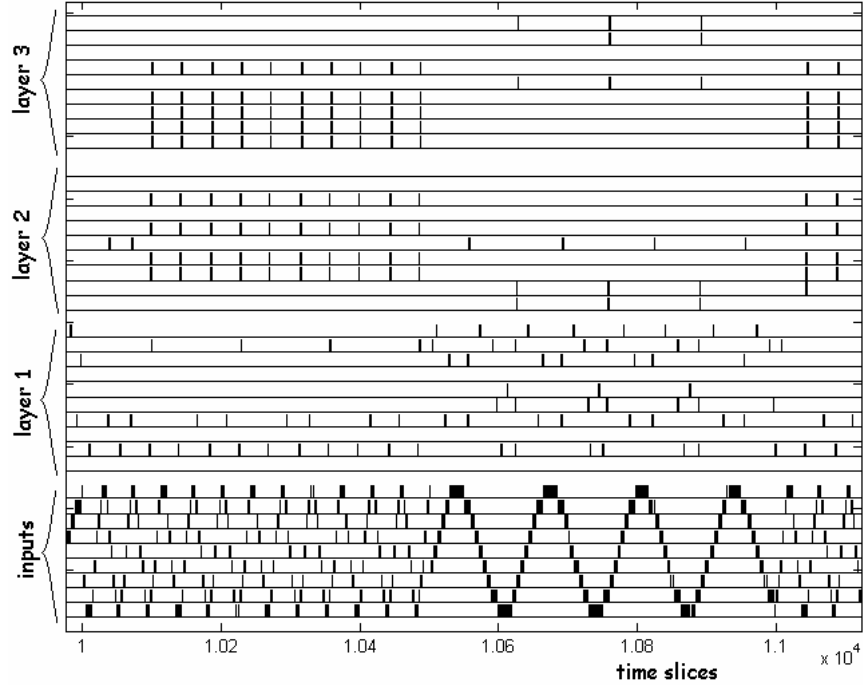


Figure 6. Neural activity on a learned frequency discriminator. The lowest nine lines are the input spikes to the network: two waveforms with periods of 43 and 133 time slices are presented. The next 10 lines show the neuron activity at layer 1, as well as the following lines show the activity of layers 2 and 3. After 10.000 time slices a clear separation can be observed at the output layer where neurons fire under the presence of only one of the waveforms.

For the simulation setup it was needed to consider some parameter ranges and resolutions, which are constrained by the hardware implementation, as shown in Table 2. Initial weights are integer numbers generated randomly from 0 to 127.

Table 2. Set-up parameters

| Neuron Parameters | | Learning Parameters | |
|-----------------------|------|----------------------|-----|
| Resting-potential | 32 | Learning rate | 6 |
| Threshold-potential | 128 | Decay rate | 4 |
| After-spike potential | 18 | Weight upper bound | 127 |
| Increasing slope | 1 | Weight lower bound | -32 |
| Decreasing slope | 1 | Learning window size | 16 |
| Potential lower bound | -128 | | |

Different combinations of two signals are presented as shown in figure 6. During the first 6000 time slices the signal is swapped every 500 time slices, leaving an in-

interval of 100 time slices between them, where no input spike is presented. Then, this interval between the signals presentation is removed, and these latter are swapped every 500 time slices (as shown in figure 6).

Several combinations of signals with different periods are presented to the network. Some of the signals are correctly separated, while others are not (Table 3). Several simulations were done for each pair of periods, being easier to have satisfactory separations for a certain range of periods (from 50 to 110 aprox.). As periods are closer to these values, it is easier to find a network able to separate them. When both signals have periods above 120 or below 80 no separations were achieved by the network.

Table 3. Signal periods presented to the network. Period units are time slices.

| Period 1 | Period 2 | Separation |
|-----------------|-----------------|-------------------|
| 40 | 100 | Yes |
| 43 | 133 | Yes |
| 47 | 73 | No |
| 47 | 91 | Yes |
| 50 | 100 | Yes |
| 73 | 150 | No |
| 73 | 190 | Yes |
| 101 | 133 | Yes |
| 115 | 190 | Partially |
| 133 | 170 | No |
| 133 | 190 | No |

It must be noticed that the separable period ranges highly depends on the data presentation to the network. In our case, we are generating a spike every 3 time slices; however, if higher (or lower) frequencies are expected to be separated, spikes must be generated at higher (or lower) rates. The period range is also given by the dynamic characteristic of the neuron: after-spike potential and increasing and decreasing slopes. They determine the membrane potential response after input and output spikes, playing a fundamental role on the dynamic response of the full network.

5. Conclusions and future work

We have presented a functional spiking neuron model suitable for hardware implementation. The proposed model neglects a lot of characteristics from biological and software oriented models. Nevertheless, it keeps its functionality and it is able to solve a relative complex task like temporal pattern recognition. Since the neuron model is highly simplified, the lack of representation power of single neurons must be compensated by a higher number of neurons, which in terms of hardware resources could be a reasonable trade-off considering the architectural simplicity allowed by the model.

With the frequency discriminator implementation, the use of the sole unsupervised hebbian learning has shown to be effective but not efficient. Although solutions were found for a given set of frequencies, we consider that better solutions could be found with the available amount of neurons. While for some classification tasks it remains useful, hebbian learning results inaccurate for other applications. On further work we will include hybrid techniques between hebbian and reinforcement or supervised learning.

This work is part of a bigger project where neural networks topologies are going to be evolved in order to explore a larger search space, where adaptation is done by synaptic weight and topology modifications. Topology exploration is done using the partial reconfiguration feature of Xilinx FPGAs.

Spiking-neuron models seem to be the best choice for this kind of implementation, given their low requirements of hardware and connectivity, keeping good computational capabilities, compared to other neuron models [4]. Likewise, layered topologies, which are among the most commonly used, seem to be the most suitable for our implementation method. However, other types of topologies are still to be explored.

Different search techniques could be applied with our methodology. Genetic algorithms constitute one of the most generic, simple, and well known of these techniques but we are convinced that it is not the best one: it does not take into account information that could be useful to optimize the network, such as the direction of the error. An example of this could be found on [5] where growing and pruning techniques are used to find the correct size of a network.

References

1. W. Gerstner, Kistler W. Spiking Neuron Models. Cambridge University Press. 2002.
2. S. Haykin. Neural Networks, A Comprehensive Foundation. 2 ed, Prentice-Hall, Inc, New Jersey, 1999.
3. A. L. Hodgkin, and A. F. Huxley, (1952). A quantitative description of ion currents and its applications to conduction and excitation in nerve membranes. *J. Physiol. (Lond.)*, 117:500-544.
4. W. Maass, Ch. Bishop. Pulsed Neural Networks. The MIT Press, Massachusetts, 1999.
5. A. Perez-Urbe. Structure-adaptable digital neural networks. PhD thesis. 1999. EPFL. http://lslwww.epfl.ch/pages/publications/rcnt_theses/perez/PerezU_thesis.pdf
6. E. Ros, R. Agis, R. R. Carrillo E. M. Ortigosa. Post-synaptic Time-Dependent Conductances in Spiking Neurons: FPGA Implementation of a Flexible Cell Model. Proceedings of IWANN'03: LNCS 2687, pp 145-152, Springer, Berlin, 2003.
7. S.M. Trimberger. Field-Programmable Gate Array Technology. Kluwer Academic Publishers, 1994.
8. A. Upegui, C.A. Peña-Reyes, E. Sánchez. A Functional Spiking Neuron Hardware Oriented Model. Proceedings of IWANN'03: LNCS 2686, pp 136-143, Springer, Berlin, 2003.
9. A. Upegui, C.A. Peña-Reyes, E. Sánchez. A methodology for evolving spiking neural-network topologies on line using partial dynamic reconfiguration. Submitted to International Congress on Computational Intelligence (CIIC'03). Medellín, Colombia.
10. Xilinx Corp. www.xilinx.com
11. X. Yao. Evolving artificial neural networks. Proceedings of the IEEE, 87(9):1423-1447, September 1999.