Neural Networks 121 (2020) 366–386

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

Design Space Exploration of Hardware Spiking Neurons for Embedded Artificial Intelligence

Nassim Abderrahmane [a,*], Edgar Lemaire [a,b], Benoît Miramond [a]

[a] Université Côte d'Azur, CNRS, LEAT, France
[b] Thales Research Technology / STI Group / LCHP, Palaiseau, France

ARTICLE INFO

Article history:
Available online 26 September 2019

Keywords:
Artificial intelligence
Neuromorphic computing
Spiking neural networks
Neural coding
Embedded systems
Power consumption

ABSTRACT

Machine learning is yielding unprecedented interest in research and industry, due to recent success in many applied contexts such as image classification and object recognition. However, the deployment of these systems requires huge computing capabilities, thus making them unsuitable for embedded systems. To deal with this limitation, many researchers are investigating brain-inspired computing, which would be a perfect alternative to the conventional Von Neumann architecture based computers (CPU/GPU) that meet the requirements for computing performance, but not for energy-efficiency. Therefore, neuromorphic hardware circuits that are adaptable for both parallel and distributed computations need to be designed. In this paper, we focus on Spiking Neural Networks (SNNs) with a comprehensive study of neural coding methods and hardware exploration. In this context, we propose a framework for neuromorphic hardware design space exploration, which allows to define a suitable architecture based on application-specific constraints and starting from a wide variety of possible architectural choices. For this framework, we have developed a behavioral level simulator for neuromorphic hardware architectural exploration named NAXT. Moreover, we propose modified versions of the standard Rate Coding technique to make trade-offs with the Time Coding paradigm, which is characterized by the low number of spikes propagating in the network. Thus, we are able to reduce the number of spikes while keeping the same neuron's model, which results in an SNN with fewer events to process. By doing so, we seek to reduce the amount of power consumed by the hardware. Furthermore, we present three neuromorphic hardware architectures in order to quantitatively study the implementation of SNNs. One of these architectures integrates a novel hybrid structure: a highly-parallel computation core for most solicited layers, and time-multiplexed computation units for deeper layers. These architectures are derived from a novel funnel-like Design Space Exploration framework for neuromorphic hardware.

1. Introduction

Over the past decade, Artificial Intelligence (AI) has been increasingly attracting the interest of industry and research organizations. Artificial Neural Networks (ANNs) are derived and inspired from the biological brain, and have become the most well-known and frequently used form of AI. Even though ANNs have garnered a lot of interest in recent years, they stem from the 1940s with the apparition of the first computer. Subsequent work and advancements have lead to the development of a wide variety of ANN models. However, many of these models settled for theory and were not implemented for industrial purposes back then. Recently, those algorithms became competitive because of two factors: first, modern computers have reached sufficient computing performance to process ANN training and inference; second, the amount of data available is growing exponentially, satisfying the extensive training data requirements for ANNs. However, the energy and hardware-resources intensiveness imposed by computation in complex form of ANNs are not matching with another current emerging technology: IoT (Internet of Things) and Edge Computing. To allow for ANNs to be executed in such embedded context, one must deploy dedicated hardware architectures for ANN acceleration.

In this case, the design of neuromorphic architectures is particularly interesting when combined with the study of spiking neural networks. Spiking Neural Networks for Deep Learning and Knowledge Representation is a current issue (Kasabov, 2018) that is particularly relevant for a community of researchers interested in both neurosciences and machine learning. Our work is part of this approach and attempts to contribute by studying more

* Corresponding author.
E-mail addresses: nassim.abderrahmane@univ-cotedazur.fr (N. Abderrahmane), edgar.lemaire@thalesgroup.com (E. Lemaire), benoit.miramond@univ-cotedazur.fr (B. Miramond).

https://doi.org/10.1016/j.neunet.2019.09.024
0893-6080/

precisely the question of the hardware design of these models. These networks are all the more advantageous as we plan to execute them in dedicated accelerators. They then take full advantage of the event-driven nature of data flows, the simplicity of its elementary operators and its local and distributed computing and learning properties. Several specific hardware solutions have already been proposed in the literature, but they are only solutions isolated from the overall design space where network topologies are often constrained by the characteristics of the circuit architecture. We recommend the opposite approach, which consists in generating the architecture that best supports the network topology. Through this study, we therefore propose an exploration framework that makes it possible to evaluate the impact of different spiking models on the effectiveness of their hardware implementation.

1.1. Spiking neurons for inference

The recent achievements of Deep Neural Networks (DNNs) on image classification have given them the leading role in machine learning algorithms and AI research. After the first phase of offline experiments, these methods began to proliferate in our daily lives through autonomous applications close to the user. Thus, more and more applications such as smart devices, IoT or autonomous vehicles require embedded and efficient implementation. However, their initial implementation on CPU was too resource-intensive for such constrained systems. Indeed, generic sequential processors are not adapted to intrinsically parallel neural algorithms. Therefore, it became essential to deploy them onto dedicated neuromorphic systems. These architectures are designed to fit the parallel and distributed computation paradigm of ANNs, permitting their implementation in embedded systems.

ANNs could be separated into three different generations, distinguished by neural computation and neural coding. The first generation is characterized by the traditional McCulloch and Pitts neuron model, which outputs discrete binary values (‘0’ or ‘1’) (Schuman et al., 2017). The second generation is characterized by the use of continuous activation functions in neural networks forming more complex architectures, such as Boltzmann Machines (Ackley, Hinton, & Sejnowski, 1985), Hopfield Networks (Hopfield, 1982), Perceptrons, Multi-Layer Perceptrons (MLP) (Rumelhart, McClelland, & PDP Research Group, 1986) and Convolutional Neural Networks (CNN) (Krizhevsky, Sutskever, & Hinton, 2017). Finally, the third generation of neural algorithms, on which this paper is focused, is Spiking Neural Networks (SNNs). In this model, information is encoded into spikes, inspiring from neuroscience. Indeed, this neuron model mimics biological neurons and synaptic communication mechanisms based on action potentials. The information is thus represented as a flow of spikes, with a wide variety of neural coding techniques (see Section 2).

According to this neural coding paradigm, SNN processing is performed in an event-based fashion: computation is operated by a spiking neuron when and only when it receives an input spike. Without any stimulation, the neuron remains idle. Hence, computation is strictly performed for relevant information propagation, in contrast to Formal Neural Networks (FNNs), where the states of every neuron are updated periodically. Moreover, the computation is usually much simpler in spiking neurons than in formal neurons. Indeed, even though several models have been identified in neuroscience studies, in a machine learning context, spiking neurons are most often based on a simple (Leaky) Integrate and Fire (IF) model (Abbott, 1999). Let us compare IF computation rule with Formal computation rule. The computation rule for Formal Neurons is presented in Eq. (1), and the computation rule for Spiking Neurons (IF model) is shown in Eq. (2):

$$y_j^l(t) = f(s_j^l(t)), \quad s_j^l(t) = \sum_{i=0}^{N_{l-1}-1} w_{ij} * y_i^{l-1}(t) \tag{1}$$

With $y_j^l(t)$ being the output of the jth neuron of layer l, $f()$ a non-linear activation function, $s_j^l(t)$ the membrane potential of the jth neuron of layer l and w_{ij} the synaptic weight between ith neuron of layer $l-1$ and jth neuron of layer l.

$$\begin{aligned} \gamma_j^l(t) &= \begin{cases} 1 \text{ if } s_j^l(t) \geq \theta \\ 0 \text{ otherwise} , \end{cases} \\ p_j^l(t) &= \begin{cases} s_j^l(t) \text{ if } s_j^l(t) \leq \theta \\ s_j^l(t) - \theta \text{ otherwise} , \end{cases} \\ s_j^l(t) &= p_j^l(t-1) + \sum_{i=0}^{N_{l-1}-1} (w_{ij} * \gamma_i^{l-1}(t)) \end{aligned} \tag{2}$$

With $\gamma_j^l(t)$ being the binary output of the jth neuron of layer l, $p_j^l(t)$ the membrane potential of the jth neuron of layer l, and θ the activation threshold of the jth neuron of layer l.

The multiplicative operation and the non-linear function $f()$ in Eq. (1) are very resource-intensive when considering hardware implementation, whereas the simple accumulation, comparison and reset operations found in Eq. (2) are much more competitive. Hence, SNNs are much more promising for low-power embedded hardware implementations than FNNs, considering the advantages in terms of event-driven computation and resource consumption brought by the Integrate and Fire model. Other spiking models exist, such as the slightly more complex Leaky Integrate and Fire (LIF) (Liu & Wang, 2001), which implies a continuously decreasing membrane potential; or the Izhikevich neuron model (Izhikevich, 2003), which reproduces more realistic biological neuron behaviors. Other neuron models are described in Kasabov (2018), which introduces details about spiking neuron models found in literature, alongside a wide variety of learning methods in spiking domain. However, we have chosen to use the simpler IF neuron model in our work, due to increased computational cost with more complex neuron models. Moreover, the IF neuron model is already known to be sufficient for spike-based classification applications (Cao, Chen, & Khosla, 2015; Cassidy et al., 2013; Cruz-Albrecht, Yung, & Srinivasa, 2012; Merolla et al., 2011; Tavanaei, Ghodrati, Kheradpisheh, Masquelier, & Maida, 2019).

1.2. Neuromorphic hardware

In this subsection, we introduce some of the most recent SNNs hardware implementations found in the literature. Those systems consist of ASIC[1] or FPGA[2] chips, designed to simulate large numbers of spiking neurons. We give a brief description of their features, alongside energy consumption information. Those information are summed up in Table 1.

SpiNNaker

SpiNNaker (Furber et al., 2014) is a fully digital system aiming to simulate very large spiking networks in real-time, and in an event-driven processing fashion. A SpiNNaker board is composed of 864 ARM9 cores, divided into 48 chips containing 18 cores each. The memory is highly distributed, as there is no global memory unit, but one small local memory unit for each core

[1] Application Specific Integrated Circuits.
[2] Field Programmable Gate Arrays.

Table 1
Neuromorphic hardware architectures.

Work	Electronics	Technology	Online Learning	Programmability	Network	Neuron model	Input data	Application domain
NeuroGrid (Benjamin et al., 2014) — 2014	Analog/Digital	ASIC - CMOS 180 nm	yes	NGPyton	Programmable	Dimensionless model	Spikes	NeuroSciences
BrainScales (Schemmel et al., 2010) — 2017	Analog/Digital	ASIC - CMOS 180 nm	yes	PyNN	FC	exp IF	Frame-based	NeuroSciences/ Classification
Loihi (Davies et al., 2018) — 2018	Digital	ASIC - CMoS 14 nm	yes	Loihi API	Conv/FC/RNN	CUBA LIF	Spikes	LASSO/ classification
TrueNorth (Akopyan et al., 2015) — 2014	Digital	ASIC - CMoS 28 nm	no	Corelets	Conv/FC/RNN	LIF	Frame-based	Multi-object detection
SpiNNaker CMP Chip (Furber, Galluppi, Temple, & Plana, 2014) — 2010	Digital	ASIC - CMoS 130 nm	yes	PyNN	Programmable	LIF, IZH, HH	Spikes	NeuroSciences
Minitaur (Neil & Liu, 2014) — 2014	Digital	FPGA - Spartan 6 LX150	no	RTL	FC	LIF	Frame-based	Classification
ConfConvNode (Camunas-Mesa, Dominguez-Cordero, Linares-Barranco, Serrano-Gotarredona, & Linares-Barranco, 2018) — 2018	Digital	FPGA - Xilinx Spartan 6	no	RTL	Conv/Pool	LIF	DVS	DVS-based classification
Fast pipeline (Yousefzadeh, Serrano-Gotarredona, & Linares-Barranco, 2015) — 2015	Digital	FPGA - Xilinx Spartan 6	no	RTL	Conv/Pool	LIF	DVS	DVS-based classification
HFirst (Orchard et al., 2015) — 2015	Digital	FPGA - Xilinx Spartan 6	no	RTL	Conv/Pool	Complex IF	DVS	DVS-based object recognition
DYNAPS (Moradi, Ning, Stefanini, & Indiveri, 2017) — 2017	Analog/Digital	FPGA - CMoS 180 nm	no	CHP language	Conv/Pool	AdExp-IF	DVS	Classification
This work — 2019	Digital	FPGA - Cyclone V 28 nm	no	N2D2, TF, Keras	FC	IF	Frame-based	Embedded-AI classification

and a shared memory for each chip. The main feature of SpiNNaker is its efficient communication system: all the nodes are interconnected through high-throughput connections designed for small packet routing, which contain Address Event Representation (AER) spikes, i.e., the address of the transmitter neuron, the date of the emission, and the destination neuron. This communication scheme has been conceived to tolerate the intrinsic massive parallelism of the ANNs. The SpiNNaker board is programmable thanks to the PyNN interface, PyNN being a Python library for SNN simulation (Davison et al., 2009; Davison, Yger, Kremkow, Perrinet, & Muller, 2007), which provides various neuron models (LIF, Izhikevich, etc.) and synaptic plasticity rules such as STDP (Spike-Time-Dependent Plasticity) (Kheradpisheh, Ganjtabesh, Thorpe, & Masquelier, 2018; Thiele, Bichler, & Dupret, 2018). In terms of energy usage, a SpiNNaker board has a peak power consumption of 1 W.

SpiNNaker is used to implement massively parallel hardware SNNs in the literature, such as NeuCube in Behrenbeck et al. (2018), where a SNN is implemented on SpiNNaker to capture and classify spatio-temporal information from EEG (ElectroEncephaloGram). Notably, this architecture offers the possibility to pause classification process to learn new samples or classes, in an Incremental Learning (Carpenter, Grossberg, Markuzon, Reynolds, Rosen, et al., 1992; Polikar, Upda, Upda, & Honavar, 2001) fashion, which is an interesting property.

Configurable event-driven convolutional node

The authors in Camunas-Mesa et al. (2018) proposed a configurable event-driven convolutional node with rate saturation mechanism in order to implement arbitrary CNNs on FPGAs. The designed node consists of a convolutional processing unit formed by a bi-dimensional array of IF neurons and a router allowing to build large 2D arrays dedicated for ConvNets inference. In this structure, each node is directly connected to four other neighboring nodes through ports that carry bidirectional flow of events. Internally, all input and output ports are connected to a router, which dispatches events to its local processing unit or to the appropriate output port. The network described by Perez-Carrasco et al. (2013) for high-speed poker symbol recognition was implemented on ®Xilinx ®Spartan 6 FPGA. With more than 5 K neurons and 500 K synapses, the generated circuit occupied 21,465 slices, 38,451 registers and 202 of block RAMs. The slower versions of the architecture showed recognition rates around 96% when all the input events were processed by the network, while less than 20% of the events were processed at real time, obtaining a recognition rate higher than 63% with a power consumption of

7.7 mW when the stimulus was being processed at real time, and even lower consumptions for slower processing: 5.25 mW when it was 10 times slower, and 0.85 mW for a slow-down factor of 100.

Conv core

This paper (Yousefzadeh et al., 2015) proposes a pipe-lined architecture for processing spiking 2D convolutional layers in a fully event-driven system. Indeed, this system takes asynchronous input data from a Dynamic Vision Sensor (DVS) (Delbrück, Linares-Barranco, Culurciello, & Posch, 2010; Lichtsteiner, Posch, & Delbruck, 2008), a bio-inspired vision sensor which outputs a continuous flow of spikes corresponding to brightness gradient variations in a dynamic image. This architecture benefits from the parallelism offered by FPGAs by implementing a 3-stages-pipeline, thus reaching the great performance of updating 128 pixels of the layer in 12 ns; while running on ®Xilinx ®Spartan 6 FPGA. On the same board, the implementation of a spiking convolution layer with a 128 × 128 pixel input and a 23 × 23 convolution kernel occupies 48% of logic resources and 68% of block RAM. This architecture uses the LIF neuron model, a bit more complex than our simple IF neuron. This system is adapted to asynchronous spiking input, whereas our system is adapted to conventional CCD (Charge Coupled Device) vision sensors, however we could adapt our architecture to DVS to benefit from the asynchronous input in parallel implementations (FPA architecture, see 5.2).

HFirst

HFirst (Orchard et al., 2015) is a Spiking CNN architecture. It is based on a frame-free paradigm, as it takes inputs from a DVS. HFirst's particularity is to focus on relative timing of spikes across neurons, benefiting from the continuous flow of input data. Hence, HFirst is dedicated to temporal pattern recognition, whereas our architecture is dedicated to static image recognition, and uses the accessible CCD sensor. Moreover, HFirst uses another IF neuron version which is more complex than ours, emulating physical behavior of an IF Neuron. This model uses several multiplications, which results in a more resource intensive implementation (17 DSP in HFirst versus 0 for ours). HFirst runs on Xilinx®'s Spartan® 6 FPGA, with a 100MHz clock, and consumes between 150mW and 200mW. It performs 97.5% accuracy on HFirst Cards data-set (4 classes), and 84.9% on HFirst Characters data-set (36 classes).

Minitaur

Minitaur (Neil & Liu, 2014) is an event-driven neural network accelerator dedicated to high performance and low power consumption. This is an SNN accelerator on®Xilinx®Spartan 6 FPGA board. The example LIF-based network implemented on the board performs 92% accuracy on MNIST dataset and 71% on 20 newsgroups dataset. The Minitaur architecture consists of 32 LIF-based cores dedicated to parallel processing of spikes. The input spikes arrive from a queue where they are stored as packets through USB interface. Those packets are encoded on 6 Bytes: 4 Bytes for timestamp, 1 Byte for layer index and 2 Bytes for the neuron address (Address-Event-Representation). This is a semi-parallel architecture, where some layer are processed in parallel, and some layer are processed sequentially. Minitaur achieves 19 million neuron update per second on 1.5 W of power and it supports up to 65 K neurons per board within fully-connected layers based SNN.

Loihi

Loihi (Davies et al., 2018) is again a fully-digital chip containing 128 cores, each of which is able to simulate up to 1024 different neurons. The memory is also largely distributed, with each core having a local 2MB SRAM memory unit. The chip also contains 2 × 86 cores and 16MB of SRAM synaptic memory. Accordingly, it is able to support up to 130 000 neurons and 130 million synapses. In contrast with previous systems, the Loihi board is able to perform learning. The chip can be programmed to implement various learning rules, notably STDP. The chip is able to simulate up to 30 billion SOPS, with an average of 10pJ per spike.

TrueNorth

TrueNorth (Akopyan et al., 2015) is another fully-digital system, capable of simulating up to 1 million spiking neurons. A TrueNorth board is composed of 4096 Neurosynaptic cores dedicated to LIF neuron emulation. Each core contains a 12.75 KB of local SRAM memory, and is time-multiplexed up to 256 times so that one core can simulate 256 different neurons. Similar to SpiNNaker, the communication scheme is asynchronous, event-based and able to tolerate a very high level of parallelism. TrueNorth can perform 46 billion synaptic operations per second (SOPS) per Watt, with a power consumption of 100mW when running a 1 million neurons network. The system is programmable thanks to the Corelet programming language (Amir et al., 2013), allowing to tune neuron parameters, synapse connectivity and inter-core connectivity.

DYNAPs

DYNAPs (Moradi et al., 2017) is a reconfigurable hybrid analog/digital architecture. Its hierarchical routing network allows the configuration of different neural network topologies. This interesting method also tries to solve the compromise between point-to-point communications and request broadcasting in large neural topologies. The use of mixed-mode analog/digital circuits allowed to distribute the memory elements across and within the computing modules. As a counterpart, this requires the addition of conversion circuits. The analog parts are operated in subthreshold domain to minimize dynamic power consumption and to implement biophysically realistic behaviors. The approach is validated by a VLSI design implementing a three-layer CNN network. If the circuit consumption is low (about ten pJ per data movement in the network), the implementation of the 2560 neurons of the targeted spiking CNN requires the use of a PCB composed of 9 circuits. The overall consumption and scalability of the approach therefore remains to be confirmed.

BrainScaleS

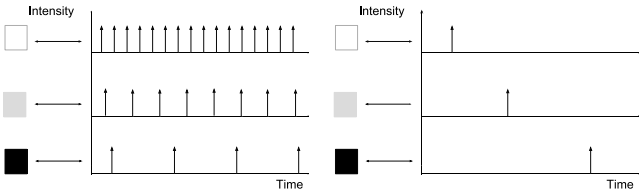
BrainScaleS (Schemmel et al., 2010) is a mixed digital–analog system. The processing units (neuron cores) are analog circuits, whereas the communication units are digital. BrainScaleS implements the adaptive exponential IF neuron model, which can be configured to reproduce many biological firing patterns. BrainScaleS is composed of HiCANN (High-Input Count Analog Neuronal Network) chips, which are able to simulate 224 spiking neurons and 15 000 synapses. Several HiCANN units can be placed on a wafer, so that a single wafer can simulate up to 180 000 neurons and 40 million synapses. The system also integrates general purpose embedded processors, which are able to measure relative spike timings, thus plasticity rules such as STDP can be implemented. Other plasticity rules can also be programmed, and a PyNN interface allows users to program the network in a similar fashion to SpiNNaker. The BrainScaleS platform consumes between 0.1 nJ and 10 nJ per spike depending on the simulated network model, and reaches a maximum of 2 kW of peak power consumption per module.

NeuroGrid

NeuroGrid (Benjamin et al., 2014) is also a mixed digital–analog system, which targets real-time simulation of large SNNs. It employs subthreshold circuits, to model neural elements. The synaptic functions are directly emulated thanks to the physics of the transistors operating in the subthreshold regime. The board is composed of 16 NeuroCore chips, interconnected by an asynchronous multicast tree routing digital communication system. Each core is composed of 256*256 analog neurons, so that NeuroGrid is able to simulate up to 1 million neurons and billions of synaptic connections. Concerning energy, NeuroGrid consumes an average of 941 pJ per spike and has a peak power consumption of 3.1 W.

1.3. Contributions

The hardware accelerators presented so far are largely destined to conduct large-scale simulations of brain-like neural networks, with a bio-mimetic implementation offering several neuron and synapse models (Table 1). Therefore, they are either designed for general purpose simulation of bio-inspired neural models, or for processing data coming from event-based cameras. They are not easily programmable from classical machine learning frameworks. This enables us to combine the efficiency of unsupervised learning and the efficiency of spiking neurons applied to prevalent frame-based sensors. Indeed, in embedded AI applications, the solution has to offer state-of-the-art prediction accuracy. Previous work (Khacef, Abderrahmane, & Miramond, 2018) has shown that SNNs cost about 50% less in terms of hardware, while having approximately the same accuracy compared to MLP (FNNs). In other words, mapping a traditional neural network to a spiking one does not severely impact the recognition rate (Diehl et al., 2015; Perez-Carrasco et al., 2013), and results in more economical hardware. Therefore, in this paper we adopt the same approach, consisting in transcoding FNNs to SNNs to solve classification problems. Such an approach has been studied in Diehl et al. (2015) and Perez-Carrasco et al. (2013) but few studies have explored the impact of spike coding on both performance and power efficiency (Cao et al., 2015; Luo, Wan, Liu, Harkin, & Cao, 2018). Indeed, our network is first trained in formal domain, and its weights are then exported to be used in an SNN with the same topology, which is then directly ready for inference. Note that there exist learning methods directly in spike domain, such as SpikeProp or STDP (Kheradpisheh et al., 2018; Mostafa, 2018; Mozafari, Ganjtabesh, Nowzari-Dalini, Thorpe, & Masquelier, 2018; Thiele et al., 2018). Additional information concerning spiking learning methods is available in Kasabov (2018), which presents a complete survey of Spiking Neural Network training techniques. In this paper, we are dealing with supervised feed-forward networks trained with back-propagation learning, because they are the most dominant deployed models when considering hardware integration (Sze, Chen, Yang, & Emer, 2017; Tavanaei et al., 2019). In our study we will also focus on neural coding, spike generation and their impact on neuromorphic system efficiency. Our intuition is that using Time Coding rather than Rate Coding, as is widely used in related studies, leads to a system with reduced power consumption. When transcoding an image in spike domain with Time Coding, for example, each pixel will fire at most once (Fig. 1(b)). With rate coding, however, spike trains are emitted for each input pixel 1(a), which results in a greater activity in the network, thus increasing resource and energy intensiveness of the system. Therefore, we developed innovative spike coding method based on Time Coding paradigm, such as Spike-Select and First-Spike. In the context of hardware SNN implementation, we have also developed NAXT (Neuromorphic Architecture eXploration Tool), which is a software for high-level neuromorphic system simulation. Developed in SystemC (Panda, 2001), this software provides coarse energy consumption, latency and chip surface estimations for various built-in architectural configurations. Thus, NAXT acts as the first evaluation tool in our architectural exploration of neuromorphic systems for fast but coarse evaluation of architectural choices. More details will be given in Section 4. We have chosen to develop our own high-level hardware SNN simulator, as existing simulators did not allow to perform hardware estimations with such high-level description, which makes our simulator innovative. High-level exploration with NAXT provides information on the suitable architectural choices, such as parallelism and memory distribution. From those results, architectures are built in VHDL to be tested at Register Transfer Level, thus furnishing precise timing, logic resources and energy measures. Notably, we propose a novel Hybrid Architecture, which combines the advantages of both multiplexed and parallel hardware implementations.



(a) Rate Coding paradigm (b) Time Coding paradigm

Fig. 1. Neural coding methods.

1.4. Outline of the paper

The remainder of this paper is organized across four sections. First, we deal with methods for encoding information into Spikes in Section 2. Novel neural coding methods are described and evaluated alongside common Rate and Time coding methods. In the four next parts of the paper, our funnel-fashioned Design Space Exploration framework is described in chronological order: In Section 3, we give a brief description of the framework's philosophy, alongside preliminary steps of our Design Flow. In Section 4, we deal with the high-level exploration step, describing the novel NAXT (Neuromorphic Architecture eXploration Tool) software and its simulation results on a typical SNN for MNIST classification. Section 5 is dedicated to low-level implementation of SNN hardware architectures based on NAXT results, and their RTL (Register Transfer Level) simulation information. An innovative hybrid architecture with both parallel and multiplexed computation cores will be introduced and evaluated. Lastly, we discuss the work while presenting some perspectives in Section 6, and conclude the paper in Section 7.

2. Neural coding

In this section, we are going to focus on neural coding methods: those are the different ways in which information can be encoded into spikes. Existing and novel coding methods will be presented.

2.1. Rate Coding versus Time Coding

In SNN architectures, information is encoded in spikes. The spikes, also called "action potentials" or "nerve pulses" in biology, are generated by a spiking neuron, in a process called "firing". In a feed-forward SNN with Fully-Connected (FC) layers, these spikes are transmitted to all the neurons of the next layer. Several neural

coding methods have been proposed by neuroscientists, including Rate Coding, Time Coding, Phase Coding, Rank Coding, Population Coding, etc. Brette (2015). In this study, we focus on Rate Coding and Time Coding for two reasons:

1. Rate Coding: for its maturity. When using this method, SNNs reach State-of-the-Art performance on classification applications (Cao et al., 2015; Khacef et al., 2018);
2. Time Coding: when used, fewer spikes are propagated in the SNN, which reduces computation and resource intensiveness during inference (Mostafa, 2018; Yu, Tang, Tan, & Yu, 2014).

Based on these methods, we propose some modified versions of the standard Rate Coding to make trade-offs with the temporal coding paradigm: maintain high accuracy and reduce the number of spikes flowing in the network. Indeed, the energy consumption of an SNN hardware implementation is directly proportional to the number of spikes it generates. As mentioned in Cao et al. (2015), an estimation of the energy consumed by processing an image is calculated using Eq. (3).

$$E_{total} = N_{spikes/image} * \alpha \qquad \text{(J/image)} \tag{3}$$

Where E_{total} the average energy consumed for the processing of an image, $N_{spikes/image}$ the total number spike emission per input pattern, and α the energy consumption of a spike emission. Note that the spiking-activity-related energy consumption varies from one accelerator to another and obviously depends on the specific architecture. In this paper, we consider three different amounts: α_FPA, α_TMA and α_HA related to the three architectures that will be described in Section 5.

2.1.1. Rate Coding

Rate Coding is the most widespread method for converting formal data into spike trains. The spike train's period is computed based on the formal data value following Eq. (4). In Fig. 1(a), three pixels of a gray-scale image are transformed into spike trains: each pixel is represented by a spike train whose frequency is proportional to its intensity (image processing). Note that some Rate Coding techniques such as Jittered Periodic apply a random factor to spike emission times, which increases network's prediction accuracy. Among several rate coding techniques presented in a previous work (Abderrahmane & Miramond, 2019), we are using Jittered Periodic method as it reaches the highest performances while not increasing the spiking activity compared to the other rate-based techniques.

2.1.2. Time Coding

The Time Coding method encodes information into spike emission date, which allows to use only one spike per input pixel (for image processing example). Unlike rate coding, a gray-scale image is encoded by signals holding only one spike per pixel. This latter is emitted in a time t that is inversely proportional to the pixel's intensity (Mostafa, 2018; Yu et al., 2014), as depicted in Fig. 1(b). In this model, spikes are dependent on each other, because their arrival times can be interpreted only relatively to other spikes.

Initially, we were interested in the work proposed by H. Mostafa in 2017(Mostafa, 2018), with a supervised learning algorithm based on temporal coding. The SNN processes input data that are first binarized and then transformed to the so-called Z-domain (change of variable: $exp(t) = z$), where all the computations are held. In this approach, a typical neuron has a function called *Get_Causal_Set()*, which returns a set of neurons from its previous layer participating in the process of firing (causal neurons). Indeed, this function uses all the previous layer's spiking times to deduce this group of causal neurons. Thus, if we suppose that it processes the spiking times to compute the emission time $t_{emission}$, then this latter would be greater than all the processed spiking times. Meanwhile, it is mentioned in the paper (Mostafa, 2018) that the spike emitted at $t_{emission}$ is fired after that the causal neurons have fired, but not after all the previous layer neurons. Hence, the information exchanged between neurons does not correspond to the real time that we can measure in event-based system. Therefore, from a hardware perspective, this method is not viable as the hardware implementation would not operate in real time. Thus, the methods presented in the following section seems more appropriate for hardware implementation.

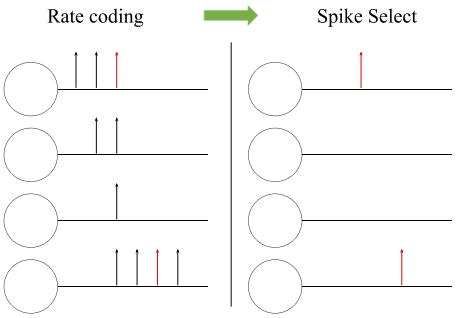


Fig. 2. Spike Select method effect on the first hidden layer neurons. The applied filter consists in raising the threshold value from 1 to 3.

2.2. Exploring novel neural coding methods

In this subsection, we will describe innovative neural coding methods we have developed.

2.2.1. Spike select

Statistical results from Rate Coding SNNs are shown in Fig. 13, where most of the spiking activity is located in the input layer. We note that only a few spikes are emitted by the deeper layers, which is sufficient for classification. In other words, most of the spikes generated by the input layer does not impact the winner class selection process. Therefore, we propose a novel kind of Rate Coding: the Spike Select method. This method consists in identifying the spikes directly involved in the classification process, by filtering them within the first hidden layer neurons. In doing so, we ensure that only the spikes intended to excite the winning class neuron are emitted. Fewer spikes will propagate to the output layer, but everyone of them will be exciting the winning class neuron. Thus, the Terminate Delta (see Part 5.1.3) procedure is still valid, even if it often takes longer to complete the Terminate Delta condition. In this regard, the latency of the whole process is increased, resulting in a higher number of spikes generated before the filter. However, the number of spikes propagating after the filter remains low. From the hardware perspective, this is a very promising method that allows for efficient hardware usage, especially with the hybrid architecture presented in Section 5.4. Indeed, in this architecture, the first hidden layer is implemented in parallel, and the deeper layers are time-multiplexed: this architectural configuration fits well with the Spike Select coding method.

Fig. 2 shows an example of some first hidden layer neurons to which the Spike Select method has been applied. The filter here consists in raising the threshold from 1 to 3, which reduces the number of emitted spikes from 10 to 2.

The value of the new raised threshold is determined by analyzing the SNN spike flow using standard Rate Coding.

2.2.2. Single Burst

With Single Burst coding method the input data stimulus is mapped to temporal domain. An input data value is represented using one spike, which is emitted at a specific time t, computed by "$t = |1 - v| * w_t$", with t the emission time, w_t the time window dedicated for the generation of the spikes, and v the input value. This method is an adaptation of the existing Single Burst stimulus type in N2D2 (Bichler, Briand, Gacoin, & Bertelone, 2017).

2.2.3. First Spike

Derived from standard rate coding, the novel First Spike method we have developed is an intermediate version between time and rate coding paradigms, having aspects in common with both methods. On one hand, as for time coding, it only uses one spike to represent information. On another hand, similar to rate coding, the compatible neuron model is the IF-neuron.

The pseudo-algorithm in Fig. 3 shows how the information (v) is converted to spike domain using the First Spike method. First, we have to define some parameters which will be used in the conversion process. A period is calculated based on value v with $p = f(v)$, using the function $f()$ in Eq. (4).

$$f(\mathbf{v}) = 1/(f_{max} + (1 - |v|) * (f_{min} - f_{max})) \tag{4}$$

Where, f_{min} and f_{max} are minimum and maximum frequency parameters. Then, *period* is used to compute the time step Δt, which corresponds to the date of the next spike emission, thanks to the function *Deviation*() presented in Eq. (5).

$$Deviation(p) = f_{Udist}(f_{Ndist}(p, s_{dev})) \tag{5}$$

With s_{dev} being the standard deviation, $f_{Ndist}()$ the Random normal distribution function and $f_{Udist}()$ the Random uniform distribution function.

Then, this Δt value is compared to *Tmin*, which is the minimum spike delay (no spike can be emitted earlier than *Tmin*). If $\Delta t > Tmin$, the spike will be emitted at time Δt, whereas if $\Delta t < Tmin$, the spike will be emitted at time *Tmin*.

This process is done only once, as this method consists of only one spike emission per input pixel.

2.3. Results

In this subsection, we present the experimental results of the different neural coding methods. We first compare the FNN accuracy results to Rate Coding based SNNs. Then, we analyze performances of the different neural coding methods. For training, validating and testing the ANNs, we have used the MNIST data-set, which is a handwritten digits database of 70 000 images (60 000 for learning and validation, 10 000 images for testing) (Lecun, Bottou, Bengio, & Haffner, 1998).

Spiking versus formal neural networks

We test the robustness of the mapping method through several ANN topologies. For this purpose, we are using N2D2 framework following the steps presented in Section 3.3.

The ANN topologies are typical MNIST recognition topologies (784 inputs, 10 outputs) with variable hidden layer sizes. Table 2 shows the accuracy results for each topology in both domains. These results are nearly the same in both ANN domains, with a small loss in the spiking domain. Thus, the mapping from formal to spiking domain does not significantly degrade the accuracy, which justifies, in part, the adoption of SNNs instead of FNNs. In Table 3, the accuracy results obtained in this paper are compared to the different SNNs that we found in literature. Indeed, we obtained slightly higher accuracy compared to those in Du et al. (2015) and Mostafa (2018). However, with 1500 fewer neurons than in Diehl et al. (2015), we have an accuracy loss of 0.36%.

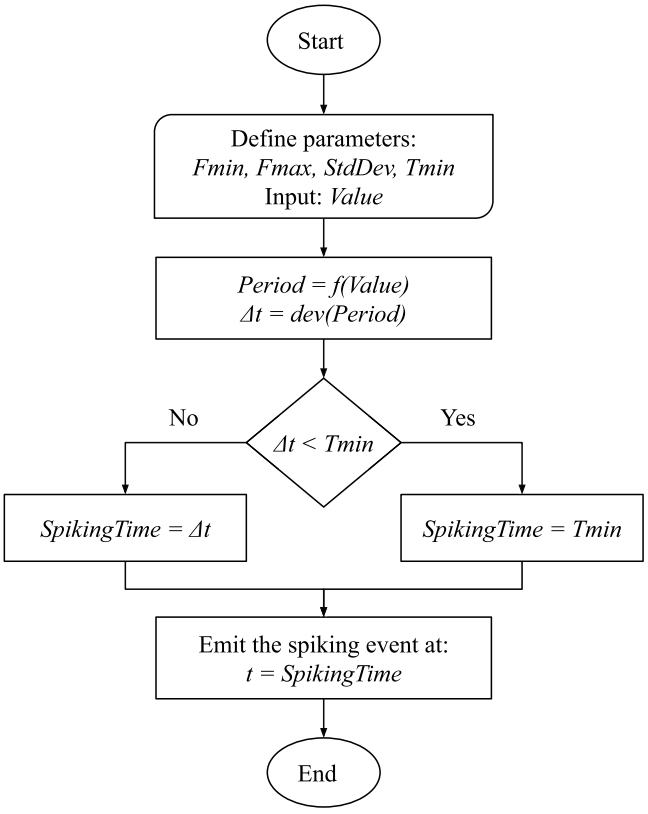


Fig. 3. First Spike method flow-chart. In this method, first, a period corresponding to the input value is computed, which is used to calculate the time step. The time step is the amount by which the actual time is increased to get the spike emission date (only one spike per input value). Fmin: minimum frequency; Fmax: maximum frequency; Tmin: spikes minimum separation time; Value: input value; Period: period equivalent to input value; f(): period conversion function; dev(): deviation function; SpikingTime: spike emission time.

Table 2
FNNs versus SNNs accuracy results on MNIST data-set. The neural coding method used the SNNs is Jittered Periodic. The results correspond to an average of 10 simulations.

ANN topology	Accuracy (%)	
	Formal	Spiking
784-100-10	96.42	96.30
784-200-10	97.44	97.29
784-300-10	97.85	97.74
784-300-300-10	98.08	98.00
784-300-300-300-10	98.35	98.24

Table 3
Classification accuracy results of different SNNs.

SNN topology	Accuracy (%)
In Diehl et al. (2015): *784-1200-1200-10*	98.60
In Khacef et al. (2018): *784-300-10*	95.37
In Du et al. (2015): *784-300-10*	95.40
In Mostafa (2018): *784-800-10*	97,55
In this paper: *784-300-10*	97.74
In this paper: *784-300-300-300-10*	98.24

Neural coding methods

We present, in Fig. 4, an illustration of the results obtained with the modified Rate Coding methods. The results are presented in histogram format showing the evolution of the number of propagating spikes in the network with respect to the neural coding method.

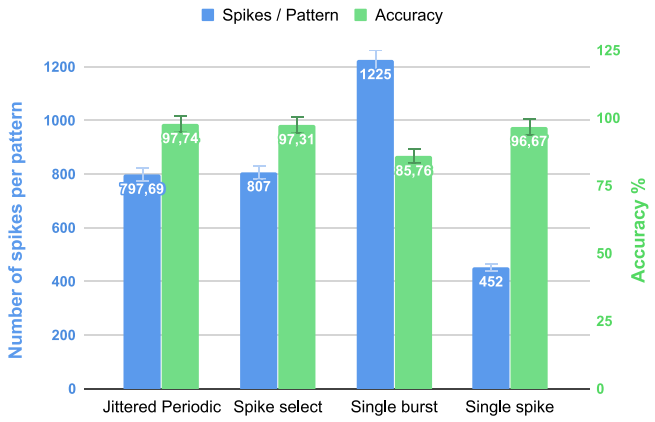


Fig. 4. Neural coding method impacts the accuracy and the number of spikes processed by the SNN in average for a pattern. The histogram represents results obtained for the 784-300-10 SNN topology using the MNIST test data.

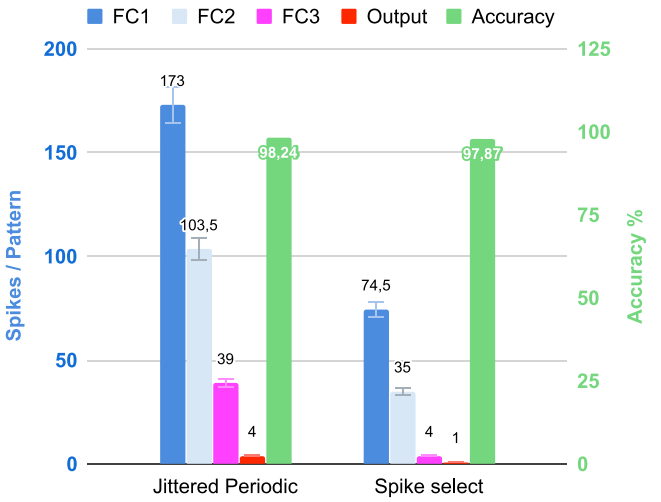


Fig. 5. Spike Select versus Jittered Periodic in terms of accuracy and number of spikes generated for a pattern in SNN deeper layers. The histogram represents results obtained for the 784-300-300-300-10 SNN topology using the test data of MNIST (cf. Table 4).

The First Spike method, due to the fact that it uses only one spike per input value, mitigates the spike throughput when compared to other methods. Meanwhile, the accuracy is kept approximately the same as with Jittered Periodic method for one hidden layer SNNs, but for the "784-300-300-300-10" deeper SNN it has a loss of 11.32%, as shown in Table 4

On an other hand, the *Spike Select* method is a well-tuned method for SNN hardware implementation despite the fact that it generates more spikes than Jittered Periodic and First Spike methods. Indeed, when looking at the distribution of these spikes over the SNN layers shown in Table 4, we observe that they are condensed in the first hidden layer and, compared to rate coding (Jittered Periodic), refer to Fig. 5, fewer spikes propagate in the deeper layers. Using this method, with only one spike in the output layer, we reach 97.87% accuracy which is very close to the Jittered Periodic equivalent (98.24%). Leaving only few spikes flowing in the remaining layers of the network, only 35% of the spikes flow in the hidden layers compared to rate coding, Spike Select is therefore well-tailored for deep SNN hardware implementations, (cf. Table 4). From this perspective, as mentioned before, the use of specific architecture with a massively parallel computation for the first hidden layer, combined with multiplexed hardware for the remaining SNN layers would be an optimized solution for the Spike Select method. Such hardware architecture will be presented in Section 5.4.

The authors in Kheradpisheh et al. (2018), proposed a Spiking Deep Neural Network (SDNN) which consists in an STDP-based CNN combined with an SVM[3] classifier. For an MNIST image, about 600 spikes are propagated over the SDNN that correspond to inhibitory events which occurred over the network. Since these events occur only in convolutional layer neurons, the input spikes (generated by DoG[4] cells) and the ones propagated in the classifier are not included in this amount of spikes. Therefore, despite the fact that SDNN spends fewer time steps compared to ours, the proposed SNN based on the Spike Select neural coding method is more efficient in terms of hardware processing, because in average only 113.5 spikes are propagated over the network (refer to Fig. 5 and Table 4). Moreover, in Diehl et al. (2015) the Rate Coding based SDNN generates from 10^3 to 10^6 spikes in the different layers for a single MNIST image.

[3] Support Vector Machine.

[4] Difference of Gaussians.

Table 4
Average number of spikes generated for processing one image by the "784-300-300-300-10" SNN with the different neural coding methods. Where, JP stands for Jittered Periodic, SS for Spike Select, SB for Single Burst and FS for First Spike.

Layer	Spikes per pattern			
	JP	SS	SB	FS
Input	724	1547	62,5	170
FC1	173	74,5	363,5	14
FC2	1035	35	1055	61
FC3	39	4	1597,5	87
Output	4	1	181,5	4
Total	1043,5	1661,5	3260	336
Accuracy %	98.24	97.87	76.80	86.92

3. Methodology for design space exploration

3.1. Description of the design flow

In this section, the adopted design flow methodology will be described. This design flow is synthesized in Fig. 6. It follows a funnel philosophy: knowing the application context, we start from a wide variety of possible hardware implementations and incrementally refine the scope to find the most suitable at the end. In our case, the example application context will be image classification.

First, a behavioral software simulation using the N2D2 framework (Bichler et al., 2017) (available online at: https://github.com/CEA-LIST/N2D2) is carried out to perform learning, test and validation for several SNN topologies with different neural coding methods. The most suitable model in terms of prediction accuracy and spiking activity (the amount of spikes processed by the SNN to perform classification inference) is selected for the following steps, and the learned parameters are extracted. A preliminary analytic study is carried to get the first estimations of flat hardware resources and memory intensiveness corresponding to the chosen SNN model: these first results will serve as a frame for the next steps of our design flow, giving indications for the most suitable architectural choices and hardware target.

Second, we perform a high-level architectural exploration aiming to confirm or invalidate the assumptions resulting from the preliminary analytic study. The NAXT simulator is configured with the model and parameters extracted from N2D2. The software will generate systemC architectures corresponding to

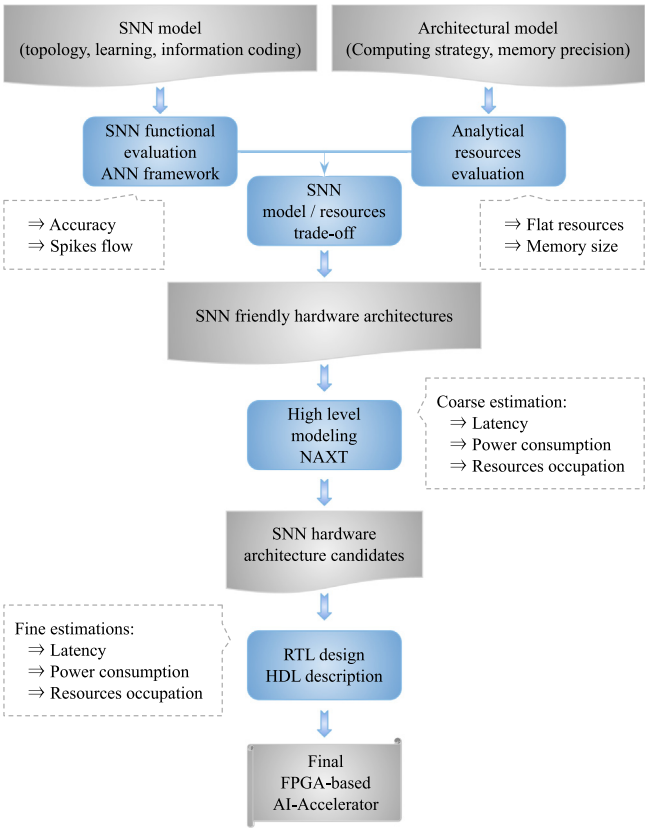


Fig. 6. Design Flow Diagram. First stage: Perform a functional evaluation of different SNN models using an ANN framework (N2D2). Simultaneously, using an analytic model evaluate the cost (memory size and flat resources) of some architectural models. Then, make a model/resources trade-off to select some SNN friendly hardware architectures; Second stage: High-level simulation of the chosen topology with different architectural choices, to select the architectural paradigm for the last stage; Third stage: Develop RTL designs of the SNN hardware architecture candidates to have finer cost estimations. Then, based on these results select one architecture as the final FPGA-based AI-Accelerator.

different high-level architectural choices, such as memory distribution, memory technology and processing parallelism. It then performs high-level simulation of their operation on the specific user-defined application task. For each simulation, we obtain coarse estimations for power consumption, surface and latency: those results allow to discriminate suitable architectural choices which will be used in following steps.

Third, a precise hardware description of the architecture is made, according to NAXT results, and using the parameters extracted from N2D2. The architecture is described in VHDL (Navabi, 1998), and a physical synthesis is performed. Thus, this last step leads to a fine-grained evaluation of a suitable architecture on FPGA (Field-Programmable Gate Arrays) or on ASIC (Application Specific Integrated Circuit).

3.2. Hardware targets of the DSE

The present work aims to deliver an architecture for Spiking Neural Network hardware implementation. To this end, two digital hardware targets are considered: Field-Programmable Gate Arrays (FPGA) and Application Specific Integrated Circuit (ASIC).

3.2.1. FPGA

In previous studies, FPGAs have been frequently employed for the design of neuromorphic computing circuits (Pani et al., 2017; Rotermund & Pawelzik, 2018). This technology can be used either for prototyping and delivering a sub-part of a greater system, or directly as the final chip design implementation. The main advantages of this technology are its high programmability, high reconfigurability, and moderate cost. As our objective is Design Space Exploration, we are interested in a reconfigurable platform: indeed, the chip must be reconfigured for each architecture. Thus, an FPGA device is an adequate technology for our purpose.

Some devices, namely SOCs (System On Chips), include one or several CPUs of the alongside with the Programmable Logic array, which offers possibilities for both software and hardware programming. As we aim to develop a general-purpose neuromorphic IP capable of executing any feed-forward SNN configuration, this type of device would suit the programmability requirement. In the present work, only Programmable Logic part has been used. However, we intend to use both in future studies, with a CPU (ARM-based) acting as a master responsible for FPGA reconfiguration and computation scheduling, and the Programmable Logic acting as a slave dedicated to inference processing.

3.2.2. ASIC

ASIC chips have also been widely employed for neuromorphic digital hardware implementations (see 1.2). In contrast with conventional processor architectures, which are designed to handle a wide variety of tasks, ASICs are fully customized to run a particular type of application. Some of those chips, such as TrueNorth (Akopyan et al., 2015; Merolla et al., 2014), are very highly specific: they are designed for one particular neuron model with very low programmability, whereas others such as SpiNNaker (Furber et al., 2014; Painkras et al., 2013), are designed with a much higher capacity for flexibility. Usually, these chip architectures are designed to support the high level of parallelism and distribution found in neural algorithms. Thus, most of the time they are based on a massively parallel computation paradigm, with great care given to the communication between computing units. However, these ASICs focus not only on pure computation acceleration, but also on the constraints of their application domain.

For integration in embedded systems for example, particular attention has to be paid to the chip surface and energy consumption limitations. These application-related constraints are also of major concern for ASIC design, and can be found in the differences between TrueNorth and SpiNNaker: the first is focused on energy savings, whereas the second is focused on flexibility. Even if the task is very similar, the implementation design is dramatically different, and so are performances: TrueNorth (Akopyan et al., 2015; Merolla et al., 2014) shows an energy consumption of 12pJ per connection, in contrast to 20nJ for SpiNNaker (Furber et al., 2014; Painkras et al., 2013). In this paper, design space exploration requires a high programmability and reconfigurability, and we have thus targeted FPGA design instead of ASIC. By this way, we favor the automatic generation architectures on a reconfigurable substrate rather than the definition of a programmable architecture on a fixed one.

3.3. N2D2 framework description

The ANN models used in this paper are learned, validated and tested using the open source Neural Network Deployment and Design (N2D2) framework (Bichler et al., 2017). This software is an event-based simulator for DNNs. A wide variety of deep-learning frameworks for design and deployment of ANNs have been described in the literature (Mozafari, Ganjtabesh, Nowzari-Dalini, & Masquelier, 2019; Parvat, Chavan, Kadam, Dev, & Pathak, 2017). However, we selected N2D2 essentially for two reasons: First, it is an open source solution that gives the ability to develop new methods without designing a whole simulator. Second, N2D2

Table 5
ANN learning hyperparameters used in this work. *LR = Learning Rate.

Hyperparameter	Value
Weight Initialization	Xavier Filler
Activation function	Linear (last layer), Rectifier (others)
Learning Rate	0.01
Momentum	0.9
Decay	0.0005
LR* Policy	Step Decay
LR* Step size	1
LR* Decay	0.993

offers the possibility to transcode and test ANNs into spiking domain, which is essential for our purpose. In order to perform simulations of SNN with N2D2, we followed these steps:

1. Define FNN topology;
2. Learn, validate and test the defined FNN;
3. Define a transcoding method to generate the SNN;
4. Test the SNN defined in 3.

Note that our configuration parameters are listed in Table 5. We have chosen Xavier Filler as a Weight Initialization method as it is a popular method which offers state-of-the-art performance (Glorot & Bengio, 2010). Moreover, we have chosen to implement Rectifier activation function (Krizhevsky, Sutskever, & Hinton, 2012; Nair & Hinton, 2010) in our hidden layers as this model offers state-of-the-art classification performance according to literature (Krizhevsky et al., 2012). Usually, in ANNs, a Softmax layer is used at the output for classification purpose. However, Softmax layers are difficult, if not impossible to implement in spike domain (Rueckauer, Lungu, Hu, & Pfeiffer, 2016). For this reason, we replace this classification method by a Linear activation function in the output layer (Diehl et al., 2015), completed by a Terminate Delta procedure to determine the winning class (see Part 5.1.3).

Once the simulation is complete, if the network prediction accuracy is satisfying, the network parameters are ready to be extracted for use in the following steps of the Design Space Exploration.

3.4. Analytic preliminary work

In this subsection, we will present some preliminary results that will drive our further investigations. These results deal with on-chip memory capacity and resource restrictions, which have to be taken into account upstream. Indeed, both of these restrictions will have strong influence on our future choices in terms of architectural model and hardware target.

3.4.1. Memory capacity: a limiting factor

On-chip memory capacity will always be limited, no matter which target hardware we choose. Indeed, even if the most recent FPGA devices such as Xilinx® Virtex® Ultrascale™ + and Intel® Stratix® 10 reach huge on-chip memory capacity, it remains inadequate to deal with most of the neural network models. For information, state-of-the-art classifiers such as VGG16 require a total of 230 MB for weight storage (Simonyan & Zisserman, 2014).

Consequently, we investigated the evolution of the required memory capacity with respect to the number of implemented synaptic connections. Our analytical model for memory capacity is based on the total memory footprint of network parameters, in our case: synaptic weight storage. Hence, our analytical approach is related to the parameters coding precision: in our case, we have chosen an 8 bits precision, as it offers a good trade-off between memory footprint and prediction accuracy; but our results can be generalized to other parameters-coding precision with a simple

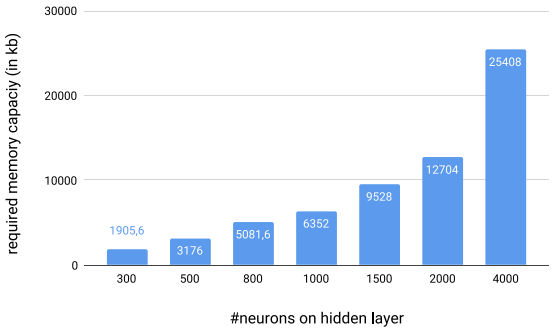


Fig. 7. Required memory capacity for a 3 layers classifier with respect to the number of neurons in the hidden layer. Synaptic weights are coded on 1 Byte each. The network has 784 input neurons, and 10 output neurons (typical ANN for testing on MNIST database).

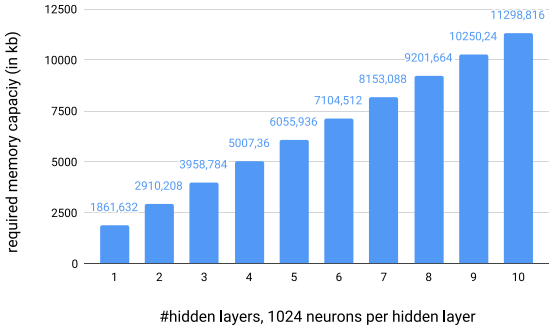


Fig. 8. Required memory capacity for a n-layer-classifier, 1024 neurons per hidden layer, with respect to the number of hidden layers. Note that, 1 Byte is used to encode 1 weight and there are 784 input and 10 output neurons.

cross product calculation. The analytical results are represented in Figs. 7 and 8. Fig. 7 depicts the evolution of required memory capacity for a 3-layer-spiking-classifier, with respect to the number of neurons in the hidden layer. On the other hand, Fig. 8 depicts the evolution of required memory capacity for an n-hidden-layer-classifier, with respect to the number of hidden layers (each hidden layer is 1024 neurons wide). Note that the memory required to store synaptic weights is the same for FNNs and SNNs: the transcoding method presented in Section 3.3 does not affect the synaptic weights.

According to the results illustrated in Figs. 7 and 8, the required memory capacity increases drastically with respect to the number of neurons, reaching several MBs for large-scale networks. The difference with our theoretical results and VGG16 memory requirement is due to the difference in number of neurons, weight coding precision, and the absence of convolutional layers in our estimations. Consequently, on-chip memory capacity is a major limiting factor for hardware SNN implementation, and has to be taken into account quite early in the design flow. Indeed, the hardware target must offer sufficient on-chip memory capacity to store model parameters. Those results also induce that intelligent memory management policy might become necessary (for example, cache hierarchies), when implementing very large models such as VGG16. Such implementation would mitigate memory footprint, even though this might result in slowing down the system and increasing logic resources intensiveness. Moreover, our model allows to evaluate the influence of weight coding precision on the memory footprint. Results are presented for a classic 784-300-10 MNIST classifier in Table 6 for various coding precision, from Binary coding to full-precision floating point (64

Table 6
Memory footprint of synaptic weight storage with respect to coding precision estimated with our analytic model.

Weight coding precision	Memory footprint
Binary (1 bit)	238 kb
Ternary (2 bits)	476 kb
8 bits (TF Lite minimum precision / Our work)	1.9 Mb
16 bits	3.8 Mb
32 bits (Half Precision Floating Point)	7.6 Mb
64 bits (Floating Point)	15.2 Mb

bits). Those results are interesting to choose a coding precision satisfying hardware target requirements, or *vice versa*.

3.4.2. FPGA Occupation: towards multiplexing

Logic resources occupation is the second limiting factor we encounter when implementing hardware SNN on FPGA devices. The FPGA occupation statistics can be obtained by FPGA synthesis simulation tools. However, this synthesis requires a long processing time, especially when synthesizing a large-scale network. In order to bypass this long processing time, we have built an analytical model capable of estimating the number of logic cells occupied on an FPGA according to the network topology and size.

To build our analytical model, we have separated a generic SNN hardware architecture in elementary modules (neurons, spike generation cell, counter, etc.). For each elementary element, we have measured corresponding hardware implementation cost in terms of logic cells, using Quartus Prime 18.1.0 Lite edition. Quite straightforwardly, every SNN topology is then expressed as a combination of those elementary modules, and hence can be related to an estimation of its flat hardware implementation cost (note that some part of the system can be multiplexed in the final design, but this model only outputs the flat hardware resources as an indicator). The results of our analytic model for a fully-parallel implementation of an SNN with 784 inputs, 10 outputs and a variable number of 100-neuron-hidden-layers, are shown in Fig. 9. As depicted in the figure, FPGA occupation grows drastically with respect to network size. Note that this model does not reproduce organization optimization performed by the synthesizer (for example, a single logic unit can be used to perform two different functions in some cases), as it can be seen in Fig. 26 which compares experimental and theoretical results. However, this model is sufficient to give a proper estimation of FPGA occupation against network size in the early stages of our design space exploration.

The analytic model shows that, consistent with our expectations, such fully-parallel implementations face FPGA capacity limits: according to the model, a fully-parallel implementation of 900 IF-neurons would cover 5465 logic cells on FPGA. Compared to real convolutional networks, which present several tens of thousands of neurons (65,000 for AlexNet (Krizhevsky et al., 2017)), it is quite obvious that the fully-parallel implementation paradigm is not viable when using FPGA devices. Moreover, when using ASIC technology instead of FPGAs, chip size would drastically grow with network size, as would production costs. Therefore, we assume that time-multiplexed architectures are way more viable when dealing with the hardware implementation of deep SNN.

On the other hand, time-multiplexing consists in implementing fewer neurons in hardware than there is in the model: each hardware neuron will thus operate successively for several neurons of the model. This method results in slowing down computation (as shown in Section 4), but allows one to implement large-scale networks with fewer resources, notably for FPGA implementation or cost reduction purposes. Those assumptions will be evaluated in further steps of our design flow (see Sections 4 and 5).

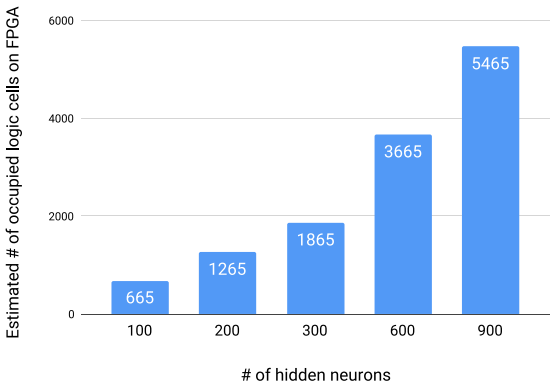


Fig. 9. Theoretical values for FPGA occupation against hidden layer size. 784 input and 10 output neurons.

4. High-level architecture exploration

In this section, we are going to introduce a tool developed by our research team, namely "NAXT" for Neuromorphic Architecture eXploration Tool, which aims to simulate SNN hardware implementations with various architectural choices, such as processing parallelism, memory distribution and memory technology. The goal is to match application specific constraints (power, consumption, logic resources) with high-level architectural choices. The simulator is configured with the SNN parameters extracted from N2D2 (topology and learned synaptic weights) and with user-defined architectural choices (i.e., level of multiplexing, level of memory distribution and memory technology). It subsequently generates a SystemC code corresponding to those parameters, and performs inference on a test data-set. The simulator estimates the chip surface, average latency and energy-consumption per inference. Hence, the role of NAXT simulator in our funnel-like architectural exploration workflow is to easily and quickly provide coarse estimations for different architectural paradigms. Although RTL modeling gives much finer results, it requires a long design and development time. Therefore, NAXT simulator is used to clear the path, as its results will guide further and finer architectural exploration in following steps of the workflow. Hence, the NAXT simulator is quite innovative as it brings hardware estimation at a very early stage of a design flow, basing on a functional description of the network. Note that the chip surface estimations are relative to an ASIC target, and are analogous to Gate Array occupation for an FPGA target. Indeed, those two metrics are relative to the same "hardware resource" evaluation: a hardware resource can be seen as a piece of circuit from the ASIC point-of-view, or as a group of logic cells from an FPGA point-of-view. Accordingly, chip surface estimations can be taken for FPGA occupation qualitative estimations.

4.1. SystemC modeling

To develop our simulator, we used SystemC (Panda, 2001), a behavioral-level hardware description library for C++. This language is often chosen for architectural exploration purposes at a high-description-level, as it enables simple functional system description, overcoming the usual finer-description-level constraints (transaction modeling, etc.). SystemC enables users to develop functional modules that run concurrent processes and communicate with each other via signals. Thus, we developed three different modules as "elementary bricks" of our hardware architecture models: a *Neural Processing Unit module*, a *Memory Unit module*, and an *Input module*. Before we start a more precise description of each module, it is important to note that our

simulator was developed according to a synchronous paradigm: every process is executed at a clock rising edge. The *clock signal* is generated by the *Input module*. Despite our enthusiasm for asynchronous processing, we have chosen a synchronous simulation paradigm for the purposes of coding ease. We plan to enable asynchronous processing simulation in future development of NAXT simulator.

4.1.1. Neural processing unit module

The *Neural Processing Unit module* (NPU) is basically a digital implementation of a spiking neuron. Thus, it is fully dedicated to the Integrate and Fire task. At every *clock rising edge*, it integrates synaptic weights corresponding to spikes received during the last cycle. The integration is done in a simple accumulator. After integration, the accumulator value is compared to the membrane's threshold value: if the threshold is exceeded, a spike is emitted at the neuron's output, and the accumulator is reinitialized. If not, it waits until next *clock rising edge* to start a new integration, and so on.

4.1.2. Memory unit module

Synaptic weights are stored in *Memory Unit modules*. Thus, *NPUs* must access *Memory Units* whenever a spike is integrated. As our architectures work on a synchronous paradigm, integration processes are run simultaneously by all *NPUs*. Consequently, a *Memory Unit* can receive several access requests at the same time, but real *Memory Units* can only answer one request at a time. Thus, our *Memory Unit* model focuses on this aspect: this module must store incoming requests in the right order, and answer those requests one by one in that same order.

4.1.3. Input module

The *Input module* is dedicated to input image transcoding. Indeed, we have to translate input data from the formal domain to the spiking domain. Various spike coding techniques exist, see Section 2. Each input image pixel is associated with an input neuron. Thus, the *Input module* is responsible for input data transcoding and spike train injection into input neurons. Ultimately, this module should disappear as we aim to simulate and evaluate a fully spiking implementation, with true spiking data coming from an asynchronous camera, for example.

4.2. Parallelism and distribution

As previously described, there are two main exploration rungs available in the NAXT simulator. The first is processing parallelism. Indeed, as SNNs are intrinsically parallel algorithms, computation parallelization should result in great acceleration of processing. On the other hand, a high level of parallelization requires a large number of *NPUs* (ideally, one per logical neuron), resulting in the drastic increase of chip surface (i.e., FPGA occupation). This first level of exploration thus allowed us to evaluate the trade-off between chip surface savings and processing acceleration. In the NAXT simulator, this exploration level is modeled by two different architectural paradigms: *Fully-Parallel Architectures*, and *Layer-Multiplexed Architectures*.

4.2.1. Fully-parallel architecture

Fully-Parallel Architecture (*FPA*) in NAXT stands for the extreme case where every logical neuron in the algorithm is implemented by an *NPU* on the chip. This architectural choice should result in fast processing but a large area.

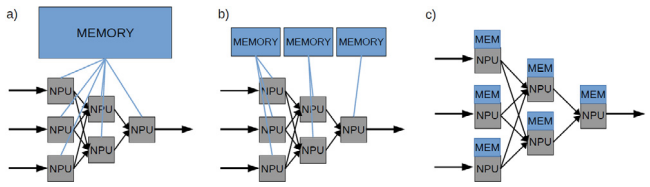


Fig. 10. Representation of all memory organizations for fully-parallel architectures: (a) centralized memory unit, (b) layer-shared memory units, (c) fully-distributed memory units.

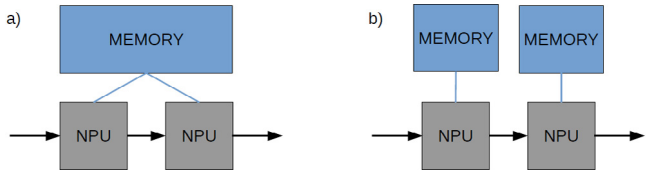


Fig. 11. Representation of all memory organizations for TMA: (a) centralized memory unit, (b) layer-shared/fully-distributed memory units.

4.2.2. Time-multiplexed architecture

Time-Multiplexed Architecture (*TMA*) in NAXT simulator stands for the case where each layer is composed of only one *NPU*. This is quite an arbitrary choice, as we could have chosen one single *NPU* for the whole network as an extreme case, but this would be the equivalent to conventional *Central Processing Unit (CPU)* architectures, which is not interesting as we want to explore innovative neuromorphic architectures. In future work, we plan to let the user choose the number of *NPUs* per layer, for flexibility and finer exploration purposes. Multiplexed architectures should result in slower processing, but will be interesting in terms of chip area savings.

4.3. Memory organization

The second rung of architectural exploration in our simulator is memory distribution. Thus, three levels of memory distributions have been developed: a *Centralized Memory* architecture (one *Memory Unit* for the whole network), a *Layer-Shared Memory* architecture (one *Memory Unit* per layer), and *Fully-Distributed Memory* architecture (one *Memory Unit* per *NPU*). Note that in the case of *TMA*, in the current version of NAXT simulator, layer-shared and fully-distributed memory organizations are the same (1 *NPU* per layer = 1 *Memory Unit* per layer in both cases).

These three different memory architectures allow users to evaluate, once again, the trade-off between processing latency, energy consumption and chip surface (i.e., FPGA occupation). For example, a *Centralized Memory* architecture will be more compact than a multitude of *Distributed Memories*, but will slow down processing as it can only answer one single *NPU* request at a time. *Layer-Shared Memory* architecture is an intermediate between both architectures. Fig. 10 depicts all memory distribution levels for fully parallel architectures, and Fig. 11 shows all memory distributions for multiplexed architectures.

4.4. Latency, power and surface estimations

The aim of NAXT Simulator is to give an estimation of power, latency and logic resources for a user defined SNN topology considering different architectural paradigms. To do so, estimations are performed *a posteriori*, using traces generated during inference simulation. More precisely, during inference, all events are recorded: spike emission, read memory access, write memory access, etc. These records, alongside with the number of clock cycles spent for processing, constitute the trace used for estimations.

4.4.1. Latency

Latency is calculated as the product of the number of clock cycles spent for processing and the clock period. The number of clock cycles being recorded in the trace file, we only have to estimate the clock period. To do so, we have chosen to constrain the clock period to the maximum memory access latency, as it is often the limiting factor in a non-pipelined architecture like ours (worst-case critical path). This latency is estimated using NVSim (Dong, Xu, Xie, & Jouppi, 2012), an open-source software aiming to simulate memory behavior for different memory technologies and technology nodes, which returns various estimations, including memory access latency.

4.4.2. Hardware resources

The Hardware resources estimation is calculated in a similar fashion than in 3.4.2: we separate our architecture in elementary modules, for each of which we measure the hardware implementation cost in terms of resources. Each architecture is expressed as a combination of elementary modules, and hence can be related to a global hardware resources cost estimation.

Note that this estimation method does not take into account placement and routing optimizations performed by FPGA design softwares (®Quartus, ®Vivado, etc.).

4.4.3. Power

The power estimation is calculated as a sum of energy consumption of the two main subparts of the system: memory and processing. Concerning memory, static and dynamic energy consumption of Memory Units are extracted from NVSim offline simulations. Static energy consumption of Memory Unis are then multiplied by the total inference latency, and summed together. Dynamic energy consumption are multiplied by the number of memory accesses (read and write), and summed together. Both of those results are summed together to give the total power estimation for memory units. Concerning Neural Processing Units, we have taken from literature (Mayr et al., 2015) the average energy consumption per spike of a state-of-the-art hardware digital spiking neuron. This average energy consumption per spike is multiplied by the number of spike emitted during inference to obtain a global power estimation of Neural Processing Units. Although this power estimation method is not directly related to our developed Neural Processing Unit architecture, it is a relevant approximation, as it concerns state-of-the-art hardware digital spiking neurons. Finally, both power estimations (for Memory Units and for Neural Processing Units) are summed together to give a global power estimation for the whole system. This power consumption evaluation method is quite approximate and thus gives coarse estimations, hence it should be improved in future works.

4.5. Results

Here, we show some data obtained with the NAXT simulator. Note that these estimations are made *a posteriori* thanks to the network activity traces (the number of spikes processed by each NPU, the number of memory accesses per memory unit, etc.). Simulations have been run for a relatively small network "784-10-10". Our simulator achieves 62% accuracy, which roughly corresponds to the equivalent N2D2 recognition accuracy for the same network. NAXT performs latency, surface and power estimations based on traces generated during processing: during each inference, we record the spiking activity of each *Neural Processing Unit*, alongside all memory accesses for each *Memory Unit*. Memory-related estimations have been computed using SRAM technology.

Results are presented in Table 7. For a better understanding of these results, they are also depicted in Fig. 12 by virtue of a qualitative cost function. This cost function is calculated as the product of three parameters (latency, energy and chip surface), as we seek to minimize those parameters at the same time. Note that this representation is purely qualitative, but gives a good indication of which architectures are the most suitable for embedded implementation.

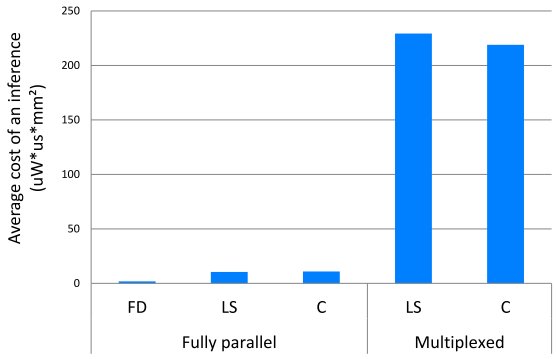


Fig. 12. Qualitative cost function for a 804 neurons hardware SNN for the different architectures available in NAXT.

Table 7
Simulation results for a 784-10-10 SNN hardware for the available architectures in NAXT, with SRAM on-chip memories.

Architecture	Fully parallel			Multiplexed	
Memory organization[a]	FD	LS	C	LS	C
Chip area (mm^2)	13	13	13	1.3	1.3
Energy consumption per inference (μJ)	3.34	3.37	3.35	27.9	27.2
Latency per inference (μs)	0.042	0.24	0.25	6.32	6.19

[a]LD: Layer Distributed; LS: Layer Shared; C: Centralized.

The obtained results are consistent with our expectations: the trade-off between chip surface on one side, and energy consumption and latency on the other side, is clearly visible in these estimations. These results show that *fully-parallel architectures* globally decrease latency and energy cost at the expense of chip surface, while *time-multiplexed architectures* have the opposite effect. This acknowledgment is quite straightforward, as *TMA* is based on an opposite design paradigm compared to parallel architectures: they are more compact, but processing serialization results in higher latency, increasing energy consumption (notably because of leakage power). Moreover, we confirmed that the more memory is distributed among processing units, the faster processing will be. Indeed, when memory is centralized, parallel access to stored data is impossible and must be serialized as explained in Section 3.4. This involves a severe increase in latency when memory is centralized. On the other hand, memory architecture does not significantly influence energy consumption and chip surface.

Therefore, we found that both multiplexed and parallel architectures have their own advantages and drawbacks, that is, the trade-off between processing latency, energy consumption and chip surface (i.e., FPGA occupation). In light of these findings, we will develop three architectures: Fully-Parallel, Time-Multiplexed, and the novel Hybrid Architecture, which uses both paradigms to optimally fit the spiking activity in the network.

Indeed, as shown in Fig. 13, in a feed-forward SNN, the number of input spikes per layer decreases drastically as we go deeper in the network: the first layers are much more solicited than

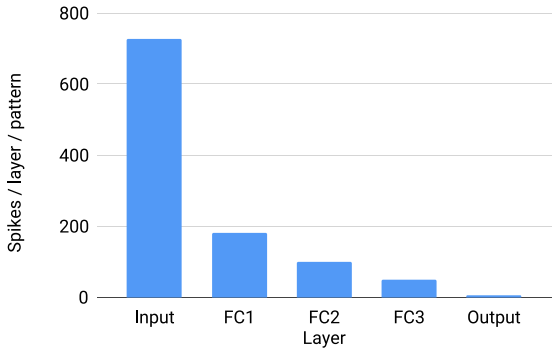


Fig. 13. Average number of spikes generated for one MNIST pattern for each layer in the "784-300-300-300-10" SNN.

deeper layers during inference. This effect is even more prominent when using our novel Spike Select neural coding method (see Section 2). Consequently, we assume that the first layers must be implemented in a fully-parallel fashion to prevent spike bottlenecks, whereas deeper layers can be implemented in a multiplexed fashion. From this assumption, a hybrid architecture has been developed in VHDL and simulated at the Register Transfer Level (RTL), which provides finer estimations than the NAXT simulator. This architecture will be presented in Section 5, alongside with *Fully-Parallel* and *Time-Multiplexed architectures*, which have also been developed and simulated at the RTL level.

5. SNN hardware architecture design

In this section, we describe the hardware design implementation of the SNN architectures studied in Section 4. Indeed, among the different models we implement: *Fully-Parallel Architecture* (*FPA*), *Time-Multiplexed Architecture* (*TMA*), and *Hybrid Architecture* (*HA*) which is the major contribution of the present work. We have selected those three different architectural paradigms according to NAXT simulation results, which enlightens how those three architectures are well suited to evaluate the trade-off between resource intensiveness, power consumption and latency. To do so, we first present the different modules used to build the different designs, then we describe the complete systems. As mentioned in Section 3.3, we use N2D2 to extract the different parameters of SNNs to move to the hardware implementation of the neuromorphic architectures. This phase is realized with the Intel® Quartus® Prime 18.1.0 Lite edition for FPGA prototyping, and ModelSim® for the validation with simulation of the design behavior.

5.1. Hardware modules

5.1.1. Integrate-and-Fire neuron module

The *IF-neuron* hardware structure is illustrated with the simplified schematic diagram in Fig. 14. In contrast to the perceptron, it does not have a multiplier and thus results in cheaper hardware with only elementary components. The module has two inputs: the input spike and its corresponding synaptic weight; and one output for output events. For clarity purpose, only positive spikes are considered.

When the neuron receives a spike, it accumulates the corresponding weight with the previous internal potential stored in a register. Afterwards, it compares this accumulated potential with the membrane potential threshold and fires whenever it is exceeded. In the case of a firing, the internal potential is decreased by the threshold amount, otherwise it remains as it is.

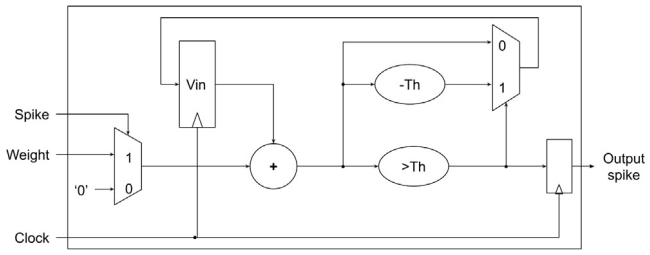


Fig. 14. IF neuron module internal structure.

5.1.2. Counter module

The *counter modules* are used for synchronization between layers and neurons. On one hand, they order the beginning and ending of computations for neuron modules and indicate the synapse addresses corresponding to input spikes (in Time-Multiplexed Architectures). On the other hand, they are linked to each other in series to ensure the coherent flow of spikes in the network and thus synchronize the different layers, referring to its usage with FPA (Fig. 20).

5.1.3. SNN class selection module

Before starting the description, let us give a quick reminder concerning class selection procedures. First of all, note that each output neuron corresponds to a data class. During inference, the winning class is selected as the most spiking output neuron. In Terminate Delta procedure, the class prediction is enacted when the most spiking neuron has spiked *delta* times more than the second most spiking neuron. On other hand, in Max Terminate, the classification process is completed whenever an output neuron (the most spiking neuron) reaches *max-value* spikes. *Delta-value* and *max-value* are user-defined parameters, usually set at 4.

For the design of our architectures, to select the output winner class we chose either *Terminate Delta* or *Max Terminate*, for which the initial software versions are defined in N2D2 framework 3.3. We have chosen those methods because they offer State-of-the-art accuracy and fast class selection. Figs. 15 and 16 show the internal structures of these modules. The input of the module is a vector (*Activations*) containing the output activity of the SNN (number of spikes emitted by each output neuron so far).

On one hand, in the *Terminate Delta* module two *maximum sub-modules* are designed to detect the maximum value of an array, which are then used to determine the winning class and to terminate the processing. The first *maximum sub-module*, namely Max1, detects the maximum value of the output activation vector, and the second, namely Max2, detects the second maximum value of this same vector. The difference between the outputs of *Max1 module* and *Max2 module* is then computed. Finally, if the difference is greater than a threshold (*delta-value*), the class corresponding to *Max1 Module* is enacted as the winner.

On the other hand, the *Max Terminate* module integrates only one *maximum* block that returns the index of the output neuron with the highest spiking activity and its activity. Then this activity is compared to a user-defined threshold *max-value*. If the maximum spiking activity is greater than *max-value*, the corresponding output neuron is enacted as the winner class, and the processing is stopped.

5.1.4. Memory modules

First-in First-out (FiFo) module

The *FiFo modules* are used in the *Neural Core (NC)*, *Network Controller* and *NPU* modules that are described later. They serve as buffers, where the output spikes of neurons are interpreted as events and are stored in a sorted way, i.e., in an ascending

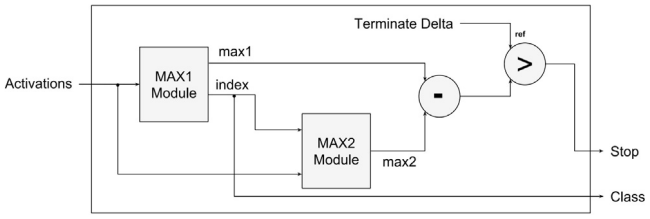


Fig. 15. Schematic diagram of the Terminate Delta module.

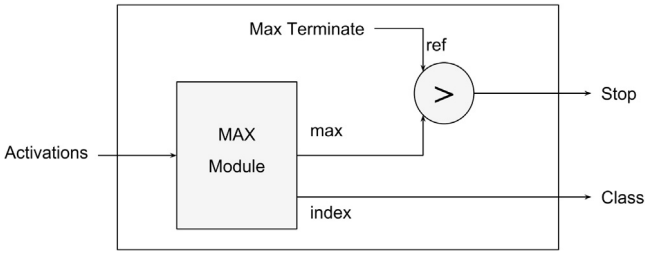


Fig. 16. Schematic diagram of the Max Terminate module.

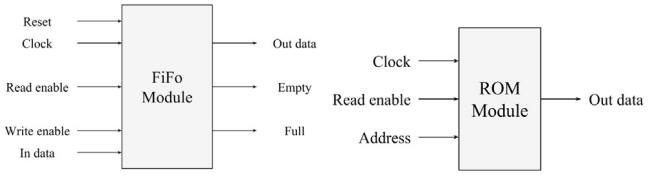


Fig. 17. FiFo and memory I/O port blocks.

order according to their times of arrival. Fig. 17(a) illustrates the schematic block of the designed module, showing its *I/O ports*. Indeed, the input and output data correspond to the *neuron address* (origin of the received spike). They are stored in this format to facilitate the search of related weights in the next layer, due to the huge number of weights. The other signals are for *read/write enable*, *clock/reset* and *FiFo memory empty/full*.

ROM module

Memory blocks are required for the proper operation of the neuromorphic system. In FPGA technology, they can be of different types: RAM, ROM, registers or latches. The *ROM modules* are used in the design of *TMA* (Time-Multiplexed Architecture), and they store the weights of the *NPU*'s logical neurons in an SNN layer. Therefore, the *ROM*s are of different sizes depending on the number of emulated synaptic connections. The *I/O ports* of a *ROM block* are shown in Fig. 17(b).

SDRAM

Memory usage is the common limitation for SNN architectures, as mentioned in Section 3.4.1, which is due to all the parameters and activities of the neurons that must be stored. From that perspective, in order to deal with deeper networks that require a significant memory size, the FPGA on-chip memory will not be sufficient. Therefore, external memory must be used to overcome this problem.

In this paper, we use *SDRAM* to reinforce the memory capabilities of the FPGA fabric. To do so, we designed a *Network Controller module* that connects the other modules to this external memory.

5.1.5. Neural Core module

The *Neural Core module* is the computation unit which emulates the two first layers (input and first hidden) of the Hybrid

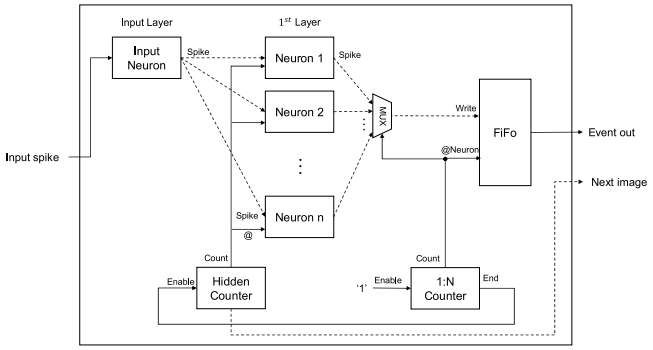


Fig. 18. Neural Core simplified schematic diagram. The input neuron forwards input spikes, spike by spike, to the 1st hidden layer neurons. The 1st hidden counter is indicating to that hidden neurons the address of the input spike in order to retrieve their appropriate weights. Another counter (1:N) is controlling a MUX component to store the 1st hidden layer's spikes in a FiFo memory.

Architecture (HA) presented in Section 5.4. This module includes an *Input Neuron Module* which forwards input spikes to downstream neurons; *IF Neuron Modules* which integrate incoming events from the *Input Neuron Module* and generate spikes according to Integrate and Fire rule. The weights are stored in registers, so that each *IF Neuron module* has its weights in a dedicated register. There are as many *IF Neuron Module* as logical neurons in the layer. Their outputs are stored in a *FiFo buffer* as events, with a *Counter Module* indicating the corresponding neuron address to be stored (see Fig. 18).

5.1.6. Neural Processing Unit module

The *Neural Processing Unit Module* (NPU) is used to emulate time-multiplexed layers. A single *IF Neuron module* will operate successively for all neurons in the layer. Moreover, the NPU includes a *FiFo Memory module*,a *Counter module* and an *NPU controller*. These modules are connected as shown in Fig. 19 to form a *NPU* which processes spiking events in a coherent way. However, besides *NPU controller*, all the other modules were presented before, and they are used by the NPU to accomplish their dedicated tasks. Consequently, only NPU controller will be described in detail. The goal of the *NPU controller* is to manage the different *NPU modules* to trigger logical neurons in a coherent way, allowing the hardware neuron to be fed with valid weights and activities.

In addition, *NPU controllers* of different NPUs are connected together in order to ensure synchronization at the network level. This synchronization is required as output classification process (Terminate Delta) depends on the arriving order of the spikes. Each *NPU module* can represent several logical neurons thanks to time-multiplexing. Note that the used weights memory type is ROM with *TMA*, and SDRAM with *HA*.

5.1.7. Network controller

The *network controller module*, used in the HA architecture, is a combination of a *FiFo module* and a *demultiplexer* (*DEMUX*); which is organized as shown in Fig. 22. The *FiFo module* accesses the SDRAM according to the *NPU* requests with a first-come-first-served policy, i.e., when an *NPU* requests a weight, this request is put in the *FiFo* queue. Then, whenever the weight is ready, it is sent via the *DEMUX block* by selecting the right *NPU*.

5.2. Fully-Parallel Architecture

This subsection describes the Fully-Parallel Architecture (FPA) we have developed for present work. This architecture has been

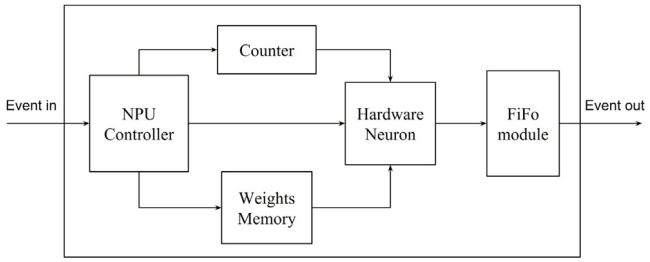


Fig. 19. Neural Processing Unit simplified block diagram. When there is an input event to process by the NPU: first, the hardware neuron is enabled by the NPU controller to retrieve the address of the logical neuron it represents from the counter and the corresponding weights from the memory block. Second, do its computation, and whenever it fires, the output spikes are stored in a FiFo as an event.

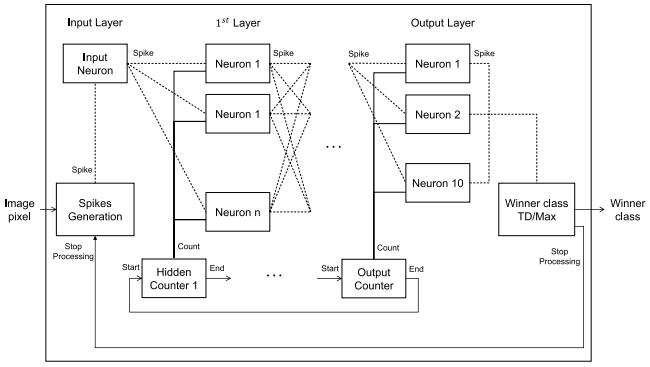


Fig. 20. FPA simplified schematic diagram. In this architecture, from the 1st layer to the output layer all neurons are implemented in hardware. Counters connected in series are coordinating the processing of the different network spikes in a coherent way.

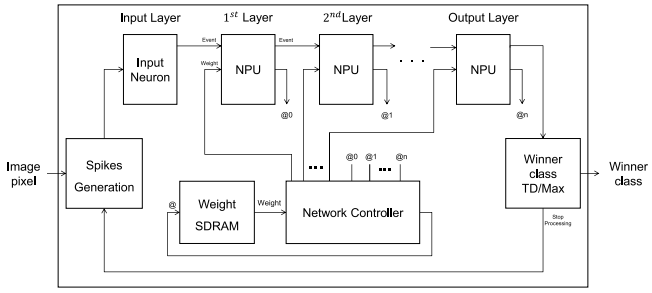


Fig. 21. TMA simplified schematic diagram. Here, the input neuron module forwards input spikes to NPUs connected in series, each NPU represents a distinct layer, that execute the SNN neurons to finally using the winner class selection module output the SNN's class.

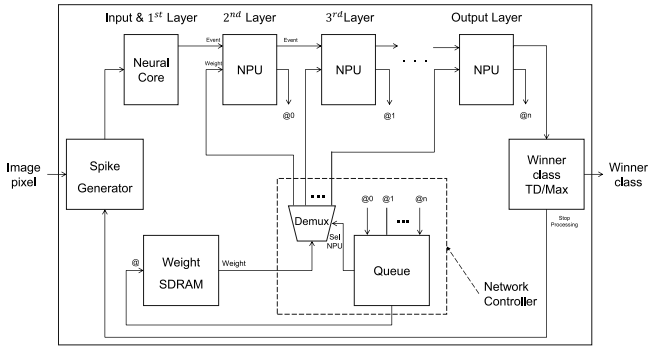


Fig. 22. Hybrid architecture simplified schematic diagram. The input and the 1st layers are implemented using a Neural Core and NPUs are used for the remaining layers (one NPU per layer). The output layer spikes are fed to the winner class selection module for classification. A Network Controller is used to manage and connect the NPUs to an SDRAM holding their logical weights.

conceived alongside Time-Multiplexed Architecture (TMA) to evaluate the trade-off between latency and resource intensiveness at a much finer level than NAXT Simulator. In the *Fully-Parallel Architecture*, all the logical neurons of the SNN are implemented in hardware. In other words, the *IF-neuron module* is instantiated as many times as the number of logical neurons. Fig. 20 shows the connectivity of the different components of the architecture. There is one *Counter module* for each layer, used to synchronize the neuron computations in the network. Indeed, in this architecture, each layer waits for the previous one to finish all its processing before starting: all spikes are processed layer by layer. The *Input Neuron* module forwards the data, spike by spike, to the first hidden layer where a *Hidden Counter* is counting them. At each clock cycle, the hidden layer *IF Neuron modules* integrate the incoming spike and store their consequent output spikes in a buffer. Then, when all the input spikes are processed, the *Hidden Counter* sends an *End Signal* to the next layer *Counter*. All the hidden layers accomplish the same process on their own incoming spikes, layer after layer. The last hidden counter enacts the end of the process to the output layer *Counter*. At this level, it is up to this counter (output layer counter) to trigger the output neurons to process the last hidden layer output spikes. Finally, the outgoing spikes are processed by the *Winner Class Selection module*, which decides whether to end the computations or to repeat the process. This fully-parallel architectural choice should result in fast processing but high logic resource intensiveness. In the following subsection, we present the second developed architecture which takes the opposite architectural choice: Time-Multiplexed Architecture (TMA).

5.3. Time-multiplexed architecture

The *TMA* architecture is designed to save hardware resources, in contrast with *FPA* architecture. In this implementation, the main computation unit is the *NPU module* described in Section 5.1.6. Contrary to *FPA*, the number of hardware neurons is smaller than the number of logical neurons: each layer is represented by one single *NPU*, instead of one *NPU* per neuron. The complete hardware architecture consists of *NPU modules*, interconnected with each other as shown in Fig. 21. As in *FPA*, the input layer is represented by a dedicated *Input Neuron module*, which forwards input spikes to the first hidden layer. Each one of the other layers is represented by one single *NPU*, which successively compute the layer's logical neurons in a time-multiplexed manner. These *NPUs* have their own *ROM memory* containing their parameters. This architecture should drastically diminish the hardware occupation, but increase the system latency as a counterpart. In other words, *TMA* and *FPA* represent the two extremes of the latency versus hardware intensiveness trade-off. In the next subsection, we will describe a middle ground between those two extremes, taking advantages from both to fit the reality of spiking activity in an SNN: the novel Hybrid Architecture (HA).

5.4. Hybrid architecture

In Section 4, it was mentioned that most of the spiking activity in the network is located in the first layer. Therefore, the first hidden layer is the most solicited layer during processing. To take advantage of this aspect, the Hybrid Architecture (*HA*) is designed, mixing both *TMA* and *FPA*. Moreover, this novel hybrid

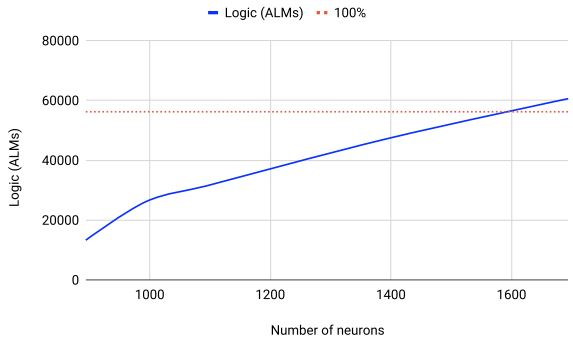


Fig. 23. FPA architecture: FPGA logic (ALM) utilization versus the SNN number of neurons; Different SNN topologies are used, see Table 8.

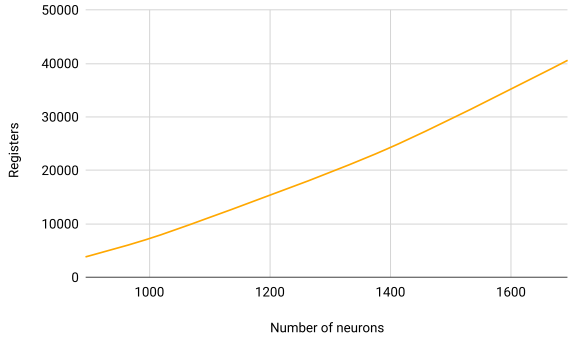


Fig. 24. FPA architecture: FPGA registers occupation versus the SNN number of neurons; Different SNN topologies are used, see Table 8.

architecture is appropriated for the use of the novel Spike Select method described in 2, in which spiking activity is concentrated in the first layer. This implementation is the main novelty of the present work, and it derives from the findings and observations we made thanks to our funnel-like Design Space Exploration framework. It is a mixture of *FPA* and *TMA*, where: first, the initial two layers are implemented using a *Neural Core* module as in *FPA*; second, the remaining layers are time-multiplexed using one *NPU* per layer, as in *TMA* . The time-multiplexed part is driven by a network controller, to retrieve the weights from the external SDRAM memory and forward them to the corresponding *NPUs*. The complete hardware schematic diagram is illustrated in Fig. 22, showing its modules and their connectivity.

5.5. Results: hardware resources occupation

In the design of AI-embedded architectures, it is important to consider resources occupation due to the lack of silicon area. Therefore, we quantify and compare the hardware cost estimations of the architectures presented in Section 5. They are described by three generic VHDL codes, which are compatible with any fully-connected multi-layer SNN topology. These VHDL codes use parameters extracted from N2D2. Their hardware costs, latency and computation performance on the *"5CGXFC7C7F23C8" Cyclone®V FPGA* board were measured through a synthesis in *Intel® Quartus® Prime Lite 18.10 edition*. Therefore, several SNN topologies of different size are implemented with the three hardware architectures.

The synthesis results using *FPA* are summarized in Table 8, giving the logic (ALM) and registers occupation related to the number of neurons. Then, those results are plotted in two graphs showing the evolution of resource intensiveness against amount of neurons (Figs. 23 and 24). From these results, we observe

Table 8
FPGA occupation (Logic ALMs and registers) of different network topologies in the FPA architecture.

SNN: Topology	Logic	Registers
784-100-10	13317	3836
784-200-10	26225	7048
784-300-10	31461	10974
784-300-300-10	47257	24008
784-300-300-300-10	60628	40600

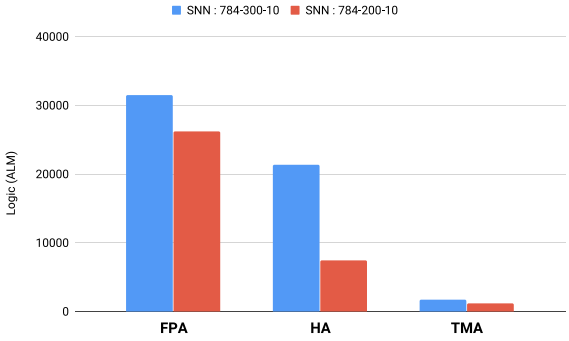


Fig. 25. Logic occupation of two SNN topologies comparing the three architectures (FPA, TMA and HA).

Table 9
FPGA cyclone V resources occupation of different SNN topologies with the TMA architecture.

SNN topology	Logic	Registers	BRAM (KB)
784-100-10	690	1255	64
784-200-10	1192	2168	128
784-300-10	1714	3082	230
784-300-300-10	3235	5937	241
784-300-300-300-10	4736	8799	249

Table 10
FPGA cyclone V resources occupation of different SNN topologies with the *HA* architecture.

SNN topology	Logic	Registers
784-100-10	2440	1383
784-200-10	7478	2434
784-300-10	21406	3455
784-300-300-10	22638	6318
784-300-300-300-10	22859	9336

that the *FPA* logic occupation is directly proportional to the SNN depth/size, i.e., increases linearly with the amount of neurons. Nevertheless, the generated circuits are supported by the FPGA fabric when the networks are smaller than the *784-300-300-10* topology, but not for bigger ones. Therefore, our first intuition regarding the limited scalability of *FPA* when used for deep SNNs is confirmed. But, we are yet to confirm if the *TMA* or *HA* architectures occupy less resources, and are thus more viable.

In this context, we have synthesized the same SNN topologies using these two architectures (*TMA* and *HA*), the results are shown in Tables 9 and 10. Different memory types and organizations are used, but the memory footprint should be the same since the same SNN topologies are implemented, i.e., equal amount of parameters and activities to store in memories. Therefore, we focus on the occupation of FPGA logic cells, where the major difference between the three architectures should be found. For improved clarity, we have plotted the histogram shown in Fig. 25 representing the logic occupation of the three architectures. As expected, *FPA* occupies much more logic resources than the other architectures.

Table 11
Latency represented as the number of cycles spent in average to process an input image by the different architectures. The results correspond to the 784-300-300-300-10 SNN using the different neural coding methods.

Coding method	Latency (cycles)		
	FPA	HA	TMA
Jittered Periodic	1039,5	84064	300540
Spike Select	1660,5	34437	496990
First Spike	332	23540	74370
Single Burst	3077	441432	459970

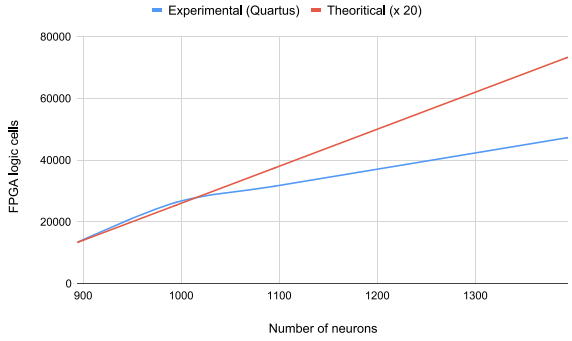


Fig. 26. FPA theoretical versus experimental FPGA occupation results with respect to the number of neurons.

On the other hand, using average spikes generated for a pattern with the same SNN topologies, we have estimated the latency of each architecture, as shown in Table 11. We observe that the processing latency is increased as hardware resources are decreased. Indeed, time-multiplexing allows to reduce the quantity of hardware resources, but relies on the sequentialization of a parallel task, thus resulting in a higher processing latency. This is why three different architectures have been designed: to evaluate the trade-off between hardware resources and processing latency. In this context, the *HA* is an intermediary solution with a significant reduction in the amount of hardware resources, while maintaining reasonable latency. On average, it has a gain of **56.19%** in terms of latency compared to *TMA* and **57.05%** in terms of logic occupation compared to *FPA*.[5]

Finally, in order to analyze the computation performance of our architectures, a measurement of SOPS (Synaptic Operation per Second) was performed on all three architectures for the 784-300-300-10 SNN topology on the same FPGA board. The *FPA* achieves the best computation performance with 51.02 billion SOPS, whereas the *TMA* only achieves 283.80 million SOPS. The *HA* is just below *FPA*, as it achieves 23.12 billion SOPS with the same topology and FPGA fabric. Their respective measured maximum computation frequencies are 83.51 MHz for *FPA*, 76.3 MHz for *TMA* and 70.95 MHz for *HA*. An ongoing work concerns the power consumption analysis of the different SNN hardware architectures.

6. Discussion

Review of our design flow

In the present study, we have presented and explained a thoughtful Design Space Exploration framework for neuromorphic hardware. This framework is based on a funnel fashion: We start with high-level modeling leading to coarse architectural choices, which will drive lower-level modeling providing finer architectural choices. Here, we will validate our design flow by showing the coherence between high-level and low-level results, and the relevance of this funnel-like design flow for neuromorphic architecture design for specific embedded applications.

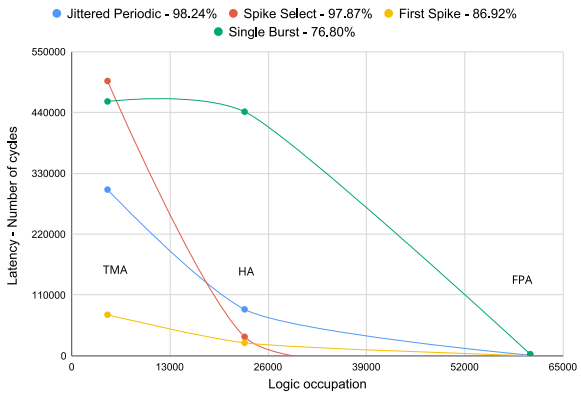


Fig. 27. Tradeoff between logic occupation and latency of the hardware architectures according to the different neural coding methods. The graph corresponds to FPGA logic occupation in Tables 8–10 and latency estimation in Table 11, which are the data recorded for the SNN of "784-300-300-300-10" topology. The obtained recognition rates on MNIST test dataset with the different methods are indicated in the legend of the figure.

As a reminder, the high-level results obtained with the NAXT simulator (in Section 4) are summarized in Table 7 and Fig. 12. As already explained, surface estimations provided by NAXT correspond to an ASIC target, and can be seen as qualitatively equivalent to logic occupation for an FPGA target. In these results, the trade-off between latency and FPGA occupation (i.e., chip surface in the figure and table) was clearly visible: *FPA* had low latency but high FPGA occupation, and the opposite was true for *TMA*. Fine-grained results provided by RTL synthesis and latency estimations for the "784-300-300-300-10" SNN are summarized in Fig. 27. This figure shows Pareto curves representing FPGA logic occupation versus latency (number of cycles) for the three hardware architectures according to the neural coding methods described in Section 2. These results are consistent with the high-level estimations, where the same trade-off can be seen between latency and logic occupation: *FPA* has a low latency but high logic occupation, whereas *TMA* has high latency but low logic occupation. If we consider the neural coding method without looking at the recognition rate, the First Spike method combined with TMA architecture has the best "latency/chip surface" trade-off. However, this method has a loss of around 10% in terms of accuracy compared to Spike Select and Jittered Periodic methods. Therefore, taking into account the accuracy criterion, the Spike Select method combined with HA architecture has the best latency/logic occupation trade-off. The method, while performing 97.87% accuracy, allows only few spikes propagating in the deeper layers of the SNN, which fits well the HA architecture making this combination one of the best choices for hardware implementation of deep SNNs.

The coherence of these results is shown in Fig. 26, which depicts the evolution of FPGA occupation (in terms of logic cells) against the number of neurons, for both theoretical estimations and Quartus® experimental results. The considered network has a fully-parallel architecture in both cases. Both curves are very similar for a low number of neurons, which confirms coherency between estimations and experimental results. The divergence observed for higher numbers of neurons is due to synthesis optimizations performed by Quartus®, which are not taken into account in our estimations. However, the two curves remain qualitatively coherent, as they follow similar linear growths. Hence,

[5] Referring to: latency Table 11 and logic occupation Tables 8–10.

the results are coherent between the high-level part and the low-level part of our design-flow.

The main interest of our Design Space Exploration framework lies in its funnel-like organization. The high-level simulations performed by the NAXT simulator are quite fast (a few seconds to a few minutes, depending on network size and architectural choices), and provide sufficiently precise results to disqualify unsuitable architectural choices. In doing so, we select a restrained number of potentially suitable architectural choices. The RTL synthesis is longer (they can take many hours for deep SNN topologies with *FPA*) but more precise, and let us determine the best choices among the pre-selected options. Indeed, operating low-level architectural exploration with RTL synthesis among the whole design space would take too long, since the range of possibilities is very wide. Thus, our presented funnel-like framework allows for efficient and reliable Design Space Exploration of Neuromorphic hardware.

Future architectures (CNN)

At this point, our work focused on fully-connected networks, the so-called classifiers (Zhang, 2000). However, this type of neural networks is restrained to simple classification tasks: they are not able to perform classification on complex data (face recognition, for example), and are not resilient to image rotation, scaling or translation. Thus, modern ANNs for complex data recognition and classification involve convolution and pooling layers: these are the Convolutional Neural Networks (CNN) (Krizhevsky et al., 2017; LeCun & Bengio, 1998). The Convolution and Pooling layers enable feature extraction and combination, resulting in a Feature Map that can be fed to a simple classifier afterwards.

In order to simulate state-of-the-art ANN hardware implementations, we aim to develop hardware architectures for spiking Convolution and Pooling layers in future work.

Asynchronous sensor: Towards frame-free SNNs for video recognition

In this work, we have focused on static image recognition. Thus, we based our approach on a transcoding method in which input data (pixel values) are translated into spikes (see Section 2). This transcoding step is one of the main drawbacks of our approach to SNNs utilization, as it may counterbalance the energy, latency and surface savings we achieved thanks to spike-based processing. When it comes to video recognition, however, this issue can be tackled by using event-based cameras.

Indeed, in contrast to static images, videos can be directly recorded in an event-based fashion, with so-called asynchronous cameras (Delbrück et al., 2010). In contrast to classical cameras, which output a succession of discrete frames, an event-based camera emits a continuous flow of events: each pixel outputs a spike whenever an edge crosses its receptive field. In other words, an asynchronous vision sensor outputs a flow of spikes representing the movement happening in its field of view. We expect that SNNs could benefit from the use of such innovative sensor, as the processing would eliminate the time-and-energy-consuming transcoding step.

Moreover, Farabet et al. (2012) have proven that such a fully event-based frame-free processing flow would bring input-to-output pseudo-simultaneity, that is, real-time processing ability. Thus, we expect that SNNs combined with asynchronous sensors would be very well suited to embedded artificial intelligence for real-time video recognition and classification. In light of these expectations, we aim to adapt our current architectures to video processing and to develop an asynchronous sensor interface for that purpose.

7. Conclusion

The present work describes an efficient novel workflow for Design Space Exploration of Neuromorphic hardware. This design flow follows a funnel-like structure: First, an analytical preliminary study determines if an architecture is feasible in terms of hardware resources (FPGA occupation) and memory footprint, which helps matching our application with a hardware target; second, a high-level exploration performed with the novel NAXT software indicates suitable high-level architectural choices; third, a low-level RTL simulation lets us determine the best implementation and provides a fine-grained evaluation of this architecture for the FPGA target. The results obtained from all these steps are consistent with the tested network models, indicating that this workflow is suitable for Neuromorphic System Design Space Exploration. In this paper, we chose the typical application case of handwritten digits recognition (MNIST dataset) to illustrate our workflow, which led to the realization of three different SNN implementations: *Fully Parallel Architecture* and *Time-Multiplexed Architecture* were developed to emphasize the two extremes of the latency versus hardware resources trade-off; and a novel and innovative *Hybrid Architecture* was created as a middle ground, deriving from the findings and observations of our Design Space Exploration work.

Moreover, the present work addresses the neural coding influence on accuracy and spiking activity. This study shows that the most suitable neural coding paradigm was the novel Spike Select coding, as it ensures high prediction accuracy and sparse spiking activity in the network. Spike sparsity implies a lower number of spikes per pattern, resulting in a shorter processing and a lower energy consumption, which is suitable for embedded system applications. Moreover, this novel spike coding method is tightly suited to our innovative Hybrid Architecture.

For deep SNNs, and according to our design flow, the most suitable architecture is our novel Hybrid Architecture, as it takes advantage of the increasing spiking activity sparsity as we go deeper into the network. This novel architecture has been developed in our lab, and to the best of our knowledge, is completely original. Combined with Spike Select Coding, it appears to be one of the most suitable approaches for future Deep SNN implementation into embedded systems.

Acknowledgments

We would like to thank Olivier Bichler and CEA LIST for providing us the N2D2 framework licence. This work is funded by "Université Côte d'Azur", "CNRS" and "Région Sud Provence-Alpes-Côte d'Azur, France".

References

Abbott, L. (1999). Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain Research Bulletin*, *50*(5), 303–304.

Abderrahmane, N., & Miramond, B. (2019). Neural coding and hardware architecture of spiking neural networks. In *Euromicro conference on digital system design (DSD)*.

Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985). A learning algorithm for boltzmann machines. *Cognitive Science*, *9*(1), 147–169.

Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., et al. (2015). Truenorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *34*(10), 1537–1557.

Amir, A., Datta, P., Risk, W. P., Cassidy, A. S., Kusnitz, J. A., Esser, S. K., et al. (2013). Cognitive computing programming paradigm: A Corelet Language for composing networks of neurosynaptic cores. In *International joint conference on neural networks*.

Behrenbeck, J., Tayeb, Z., Bhiri, C., Richter, C., Rhodes, O., Kasabov, N., et al. (2018). Classification and regression of spatio-temporal signals using NeuCube and its realization on SpiNNaker neuromorphic hardware. *Journal of Neural Engineering*.

Benjamin, B. V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J., et al. (2014). Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, *102*(5), 699–716.

Bichler, O., Briand, D., Gacoin, V., & Bertelone, B. (2017). N2D2 - neural network design and deployment. https://github.com/CEA-LIST/N2D2.

Brette, R. (2015). Philosophy of the spike: Rate-based vs. spike-based theories of the brain. *Frontiers in Systems Neuroscience*, *9*.

Camunas-Mesa, L. A., Dominguez-Cordero, Y. L., Linares-Barranco, A., Serrano-Gotarredona, T., & Linares-Barranco, B. (2018). A configurable event-driven convolutional node with rate saturation mechanism for modular convnet systems implementation. *Frontiers in Neuroscience*, *12*.

Cao, Y., Chen, Y., & Khosla, D. (2015). Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, *113*(1), 54–66.

Carpenter, G. A., Grossberg, S., Markuzon, N., Reynolds, J. H., Rosen, D. B., et al. (1992). Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps. *IEEE Transactions on Neural Networks*, *3*(5), 698–713.

Cassidy, A. S., Merolla, P., Arthur, J. V., Esser, S. K., Jackson, B., Alvarez-Icaza, R., et al. (2013). Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *International joint conference on neural networks*.

Cruz-Albrecht, J. M., Yung, M. W., & Srinivasa, N. (2012). Energy-efficient neuron, synapse and STDP integrated circuits. *IEEE Transactions on Biomedical Circuits and Systems*, *6*(3), 246–256.

Davies, M., Srinivasa, N., Lin, T., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, *38*(1), 82–99.

Davison, A. P., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, *2*, 11.

Davison, A., Yger, P., Kremkow, J., Perrinet, L., & Muller, E. (2007). PyNN: towards a universal neural simulator API in python. *BMC Neuroscience*, *8*(S2), P2.

Delbrück, T., Linares-Barranco, B., Culurciello, E., & Posch, C. (2010). Activity-driven, event-based vision sensors. In *IEEE international symposium on circuits and systems*.

Diehl, P. U., Neil, D., Binas, J., Cook, M., Liu, S.-C., & Pfeiffer, M. (2015). Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *Ieee* (pp. 1–8). IEEE.

Dong, X., Xu, C., Xie, Y., & Jouppi, N. P. (2012). Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *31*(7), 994–1007.

Du, Z., Rubin, D. D. B., Chen, Y., Hel, L., Chen, T., Zhang, L., et al. (2015). Neuromorphic accelerators: A comparison between neuroscience and machine-learning approaches. In *48th Annual IEEE/ACM international symposium on microarchitecture (MICRO)*.

Farabet, C., Paz, R., Peŕez-Carrasco, J., Zamarreno-Ramos, C., Linares-Barranco, A., LeCun, Y., et al. (2012). Comparison between frame-constrained fix-pixel-value and frame-free spiking-dynamic-pixel convnets for visual processing. *Frontiers in Neuroscience*, *6*.

Furber, S. B., Galluppi, F., Temple, S., & Plana, L. A. (2014). The SpiNNaker project. *Proceedings of the IEEE*, *102*(5), 652–665.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. (pp. 249–256).

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, *79*(8), 2554–2558.

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, *14*(6), 1569–1572.

Kasabov, N. K. (2018). *Time-space, spiking neural networks and brain-inspired artificial intelligence. (Vol. 7)*. Springer.

Khacef, L., Abderrahmane, N., & Miramond, B. (2018). Confronting machine-learning with neuroscience for neuromorphic architectures design. In *International joint conference on neural networks (IJCNN)*.

Kheradpisheh, S. R., Ganjtabesh, M., Thorpe, S. J., & Masquelier, T. (2018). STDPs-based spiking deep convolutional neural networks for object recognition. *Neural Networks*, *99*, 56–67.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (pp. 1097–1105).

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90.

LeCun, Y., & Bengio, Y. (1998). In M. A. Arbib (Ed.), *The handbook of brain theory and neural networks* (pp. 255–258). Cambridge, MA, USA: MIT Press.

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Lichtsteiner, P., Posch, C., & Delbruck, T. (2008). A 128*128 120 db 15us latency asynchronous temporal contrast vision sensor. *IEEE Journal of Solid-State Circuits*, *43*(2), 566–576.

Liu, Y.-H., & Wang, X.-J. (2001). Spike-frequency adaptation of a generalized leaky integrate-and-fire model neuron. *Journal of Computational Neuroscience*, *10*(1), 25–45.

Luo, Y., Wan, L., Liu, J., Harkin, J., & Cao, Y. (2018). An efficient, low-cost routing architecture for spiking neural network hardware implementations. *Neural Processing Letters*, *48*(3), 1777–1788.

Mayr, C., Partzsch, J., Noack, M., Hänzsche, S., Scholze, S., Höppner, S., et al. (2015). A biological-realtime neuromorphic system in 28 nm CMOS using low-leakage switched capacitor circuits. *IEEE Transactions on Biomedical Circuits and Systems*, *10*(1), 243–254.

Merolla, P., Arthur, J., Akopyan, F., Imam, N., Manohar, R., & Modha, D. S. (2011). A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm. In *IEEE custom integrated circuits conference (CICC)*.

Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., et al. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, *345*(6197), 668–673.

Moradi, S., Ning, Q., Stefanini, F., & Indiveri, G. (2017). A scalable multi-core architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *CoRR*, *abs/1708.04198*.

Mostafa, H. (2018). Supervised learning based on temporal coding in spiking neural networks. *IEEE Transactions on Neural Networks Learing Systems*, *29*(7), 3227–3235.

Mozafari, M., Ganjtabesh, M., Nowzari-Dalini, A., & Masquelier, T. (2019). Spyke-torch: Efficient simulation of convolutional spiking neural networks with at most one spike per neuron. *CoRR*, *abs/1903.02440*.

Mozafari, M., Ganjtabesh, M., Nowzari-Dalini, A., Thorpe, S. J., & Masquelier, T. (2018). Combining STDP and reward-modulated STDP in deep convolutional spiking neural networks for digit recognition. *CoRR*, *abs/1804.00227*.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. (pp. 807–814).

Navabi, Z. (1998). *Electrical and electronic technology series*, *VHDL: Analysis and modeling of digital systems*. McGraw-Hill.

Neil, D., & Liu, S. (2014). Minitaur, an event-driven FPGA-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *22*(12), 2621–2628.

Orchard, G., Meyer, C., Etienne-Cummings, R., Posch, C., Thakor, N., & Benosman, R. (2015). Hfirst: a temporal approach to object recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *37*(10), 2028–2040.

Painkras, E., Plana, L. A., Garside, J., Temple, S., Galluppi, F., Patterson, C., et al. (2013). SpiNNaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits*, *48*(8), 1943–1953.

Panda, P. R. (2001). Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on systems synthesis* ISSS '01, (pp. 75–80). ACM.

Pani, D., Meloni, P., Tuveri, G., Palumbo, F., Massobrio, P., & Raffo, L. (2017). An FPGA platform for real-time simulation of spiking neuronal networks. *Frontiers in Neuroscience*, *11*, 90.

Parvat, A., Chavan, J., Kadam, S., Dev, S., & Pathak, V. (2017). A survey of deep-learning frameworks. In *International conference on inventive systems and control*.

Perez-Carrasco, J. A., Zhao, B., Serrano, C., Acha, B., Serrano-Gotarredona, T., Chen, S., et al. (2013). Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward ConvNets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *35*(11), 2706–2719.

Polikar, R., Upda, L., Upda, S. S., & Honavar, V. (2001). Learn++: An incremental learning algorithm for supervised neural networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *31*(4), 497–508.

Rotermund, D., & Pawelzik, K. R. (2018). *Massively parallel FPGA hardware for spike-by-spike networks*. bioRxiv, Cold Spring Harbor Laboratory.

Rueckauer, B., Lungu, I.-A., Hu, Y., & Pfeiffer, M. (2016). Theory and tools for the conversion of analog to spiking convolutional neural networks. arXiv preprint arXiv:1612.04052.

Rumelhart, D. E., McClelland, J. L., & PDP Research Group, C. (Eds.), (1986). *Parallel distributed processing: explorations in the microstructure of cognition, Vol. 1: foundations*. Cambridge, MA, USA: MIT Press.

Schemmel, J., Briiderle, D., Griibl, A., Hock, M., Meier, K., & Millner, S. (2010). A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of 2010 IEEE international symposium on circuits and systems* (pp. 1947–1950). IEEE.

Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., et al. (2017). A survey of neuromorphic computing and neural networks in hardware. *CoRR*, *abs/1705.06963*.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, *abs/1409.1556*.

Sze, V., Chen, Y., Yang, T., & Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, *105*(12), 2295–2329.

Tavanaei, A., Ghodrati, M., Kheradpisheh, S. R., Masquelier, T., & Maida, A. (2019). Deep learning in spiking neural networks. *Neural Networks*, *111*, 47–63.

Thiele, J. C., Bichler, O., & Dupret, A. (2018). Event-based, timescale invariant unsupervised online deep learning with STDP. *Frontiers in Computational Neuroscience*, *12*, 46.

Yousefzadeh, A., Serrano-Gotarredona, T., & Linares-Barranco, B. (2015). Fast pipeline 128× 128 pixel spiking convolution core for event-driven vision processing in FPGAs. In *2015 International conference on event-based control, communication, and signal processing (EBCCSP)* (pp. 1–8). IEEE.

Yu, Q., Tang, H., Tan, K. C., & Yu, H. (2014). A brain-inspired spiking neural network model with temporal encoding and learning. *Neurocomputing*, *138*, 3–13.

Zhang, G. P. (2000). Neural networks for classification: a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *30*(4), 451–462.