Flexible Simulation for Neuromorphic Circuit Design: Motion Detection Case Study

Pierre Falez, Philippe Devienne, Pierre Tirilly, Marius Bilasco, Christophe Loyez, Ilias Sourikopoulos, Pierre Boulet

Compas'2017 : Parallélisme/ Architecture / Système
Sophia Antipolis, France, du 27 au 30 juin 2017

Flexible Simulation for Neuromorphic Circuit Design: Motion Detection Case Study

Pierre Falez[*], Philippe Devienne[*], Pierre Tirilly[†], Marius Bilasco[*], Christophe Loyez[‡], Ilias Sourikopoulos[‡], Pierre Boulet[*]

[*]Univ. Lille, CNRS, Centrale Lille, UMR 9189 – CRIStAL – Centre de Recherche en Informatique, Signal et Automatique de Lille, F-59000, Lille, France
[†]Univ. Lille, CNRS, Centrale Lille, IMT Lille Douai, UMR 9189 – CRIStAL – Centre de Recherche en Informatique, Signal et Automatique de Lille, F-59000, Lille, France
[‡]Univ. Lille, CNRS, UMR 8520 – IEMN – Institut d'Électronique, de Microélectronique et de Nanotechnologie de Lille, F-59000, Lille, France
Prenom.Nom@univ-lille.fr

Résumé
Les architectures neuromorphiques sont l'une des approches les plus prometteuses pour continuer l'amélioration des capacités de calcul après la fin imminente de la loi de Moore. En effet les réseaux de neurones affichent aujourd'hui des succès retentissants dans le domaine de l'intelligence artificielle et sont naturellement tolérants à la variabilité du matériel due aux processus de fabrication aux échelles nanométriques. Ils permettent en outre d'espérer un fonctionnement très économe en énergie.
Nous présentons ici une étude de cas montrant la flexibilité de notre simulateur en logiciel libre d'architectures neuromorphiques, N2S3, et les gains énergétiques potentiels de ces architectures.

Mots-clés : architecture neuromorphique, aide à la conception de circuits, simulation, réseaux de neurones à impulsions

---

1. Introduction

Neuromorphic computing has the potential to bring very low power computation to future computer architectures and embedded systems [7]. Indeed, parallel neuromorphic computing, by performing computation and storage on the same devices, can overcome the von Neumann bottleneck.

This work focuses on spiking neural networks (SNN) as they have the capability to handle natural signals and can be implemented physically through ultra-low power devices based, for instance, on CMOS [10] or memristors [6]. A comprehensive introduction and literature review about SNN was published by Paugam-Moisy and Bohte in 2012 [8]. The authors explore the computational capabilities of SNN, their learning capabilities, and their simulation.

Brette *et al.* [3] surveyed and discussed the existing work on SNN simulation in 2007. All the simulators discussed in their article as well as the more recent Brian [4] target the simulation of biological SNN. More recently, Bichler *et al.* [1] proposed Xnet, a C++ event-driven simulator dedicated to the simulation of hardware SNN. In our work, we share the goals of Xnet: "intermediate modeling level, between low-level hardware description languages and high-level neural networks simulators used primarily in neurosciences", and "the integration of synaptic memristive device modeling, hardware constraints and any custom features required for the targeted application". In addition to these goals, we put an emphasis on *flexibility* and *usability* to allow the study of various kinds of hardware designs (possibly by other researchers than us), *scalability* to simulate large hardware SNNs, and *software engineering best practices* (robust and extensible software architecture for maintainability, extensive test suite, continuous integration, open-source distribution).

We have developed N2S3 (Neural Network Scalable Spiking Simulator), an event-driven simulator dedicated to the exploration of hardware SNN architectures. The internals of N2S3 are based on the exchanges of messages between concurrent actors, mimicking the exchange of spikes between neurons. N2S3 has been developed from the ground up for extensibility, allowing to model various kinds of neuron and synapse models, various network topologies (especially, it is not restricted to feed-forward networks), various learning procedures, various reporting facilities, and to be user-friendly, with a domain specific language to easily express and run new experiments. It is available as open-source software at `https://sourcesup.renater.fr/wiki/n2s3` so that its users can share their models and experimental settings to enable others to reproduce their results. More details on the simulator are available in [2].

Flexibility in particular is a differentiating feature of N2S3. We demonstrate in this article the advantage of this flexibility by the comparison of several designs on a motion detection application. These designs differ by the topologies of their networks (recurrent or feed-forward), and their learning procedures (local or global, supervised or unsupervised, acting on the synaptic weights or delays).

Finally, since the CMOS technology is known as the most appropriate candidate for the hardware implementation of SNN, thanks to its ability for large-scale integration, we estimate the gain in terms of energy of the hardware implementation of these different neural networks based on our own ultra-low power neurons and synapses.

2. Case study: motion detection

We demonstrate the flexibility of N2S3, by using three different approachesto design a neural network to solve a motion detection task. We have decided to work on a small task, so that we can implement it on hardware without needing large numbers of neurons and synapses. The task consists in detecting the direction of the motion of a pixel on a two-dimensional grid. Each benchmark consists in a series of successive movements of a pixel on the grid, without any overlap between two inputs. Each motion has a linear trajectory, a constant velocity and a direction, and so, is represented by the successive activations of the different pixels of the trajectory.

We create two datasets, a basic one and a more complex one. For both datasets, we set the grid dimension to 10×10 to maintain a reasonable size for our networks. The simple dataset includes only four directions (up, down, left, right), no variability on the orientations of the trajectories (e.g., therefore each input has an orientation which has an angle to the x-axis of 0, $\frac{\pi}{2}$, π or $\frac{3\pi}{2}$) and only one possible velocity (0.5 pixels/ms). In order to avoid that all the trajectories pass through the center of the grid, a pixel is chosen at random as the reference point of the trajectory for each sample. Because each trajectory is parallel to the axes of the grid, all samples have exactly the same number of stimuli: 10. The complex dataset has eight possible directions (we added the diagonals), and the possible velocities range from 0.25 to 0.5 pixels/ms. To introduce some variability, we use a Gaussian distribution to generate the orientation of each trajectory. Furthermore, we apply a random orthogonal shift to each trajectory; it follows a Gaussian distribution with the grid center as its mean. We provide the option to apply a jitter noise to the data to observe the tolerance of the network to noise.

The classification score of the task is computed by taking the ratio between the number of well classified motion samples and the total number of samples. A motion is considered as well classified if the first output neuron to fire since the beginning of the motion corresponds to the class of the current motion. To assign a class to each output neuron, we select the class to which the neuron reacts most.

Thanks to the flexibility of N2S3, we can test and compare several approaches to solve this task. An interesting property of motion detection is that not only synaptic weight learning is important: synaptic delays play an essential part in the resolution task. According to the incoming temporal pattern, delays allow to synchronize the post synaptic spikes. Thus, neurons will fire only when some specific input patterns arise [5]. We are therefore also interested in setting and learning the synaptic delays. We are using several topologies and several learning processes. We retain three approaches: reservoir computing with both weight and delay learning, a small feed-forward network with global unsupervised learning of delays coupled to supervised learning of weights, and a feed forward network with no training (i.e., all weights and delays are manually set to solve the task).

3. Comparison of Three Approaches

3.1. Reservoir Computing Approach

Our first approach to solve this motion detection task is inspired from [9], which uses reservoir computing and supervised delay adaptation to learn two different patterns. The basic principle of reservoir computing is to couple a recurrent, randomly connected, layer (i.e., the reservoir), with a linear classifier. Only the classifier needs to be trained in order to map the state of the reservoir to a class (see Figure 1). More details about the network topology and the learning algorithm are available in [9].

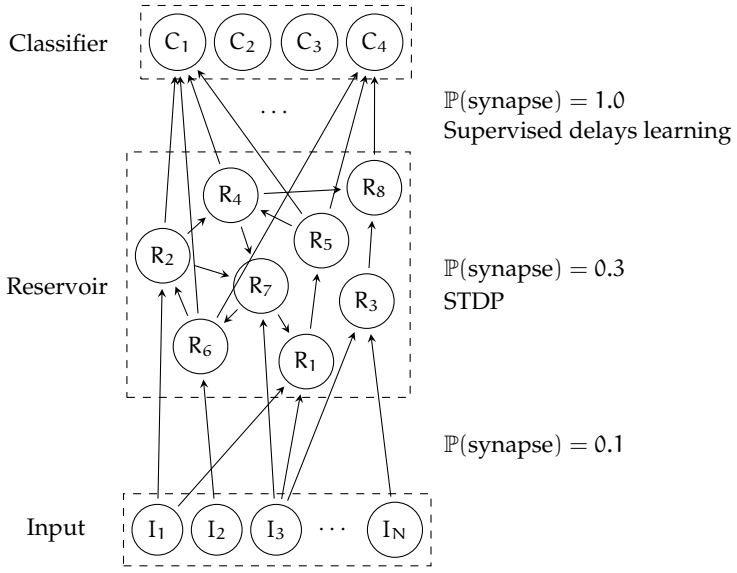

Figure 1: Topology used in the reservoir computing approach.

We study the influence of the reservoir size on the classification score. With a larger reservoir, the network yields a higher classification score: This make sense since a higher number of neurons means more states to classify the current input (see Figure 2). Our second study is about the impact of STDP on the reservoir state. After comparing results yielded by reservoirs with or without STDP, we observe that this mechanism may improve slightly the classification of the reservoir state. STDP reinforces recurrent patterns and reduces the chances of firing on an unusual pattern. This leads to better classification results by the readout.

3.2. Trained Feed-Forward Approach

The second approach is an application-specific feed-forward network. We create the smallest topology possible that can perform well on this task. This topology is a compound of three layers. The first layer reduces the dimensionality of the inputs. The two orientations, vertical and horizontal, are each mapped to a different sub-network. The second layer recognizes the input velocity, in every orientation. We have chosen ten velocity classifiers per trajectory (five per direction). Finally, the third layer classifies the directions by adjusting its synaptic weights (see Figure 3).

We use a global unsupervised method to adapt the synapse delays in the second layer. Supervised weight learning is used in the third layer to map each velocity to the correct direction.

Such an application-specific network has the advantage of achieving a better score with fewer neurons (see Figure 2), but has to be designed specifically for a given task.

3.3. Fixed Feed-Forward Approach

The third approach is a fixed network. We set all the synaptic weights and delays at the creation of the network, and so, do not train the network at all. Since the learning algorithms of SNN are not mastered yet, pre-computing the parameters can have the advantage to produce a network suited better to solve the task.

We create a two-layer feed-forward network. The first layer aims to cover a maximum of possible cases that can arise from the input of the benchmark. The neurons of the first layer each cover a

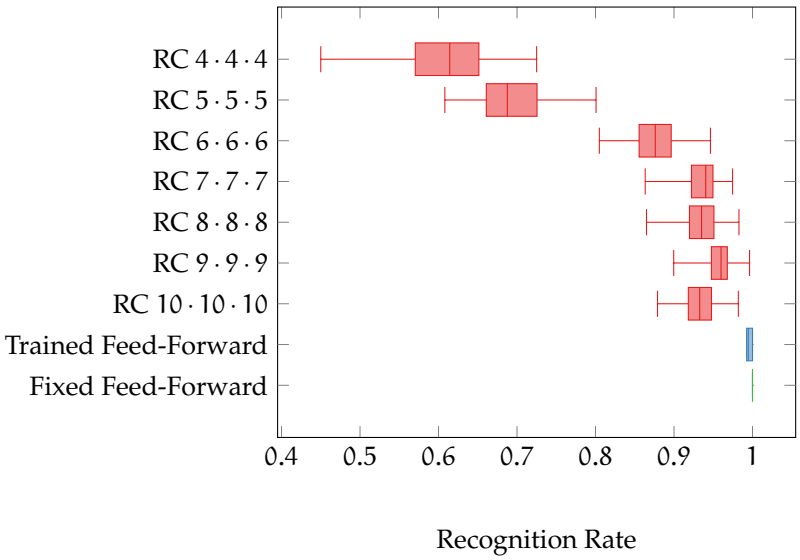

Figure 2: Result of classification with the different approaches. Each configuration is run 100 times. For reservoir computing, the network topology is regenerated randomly every time.

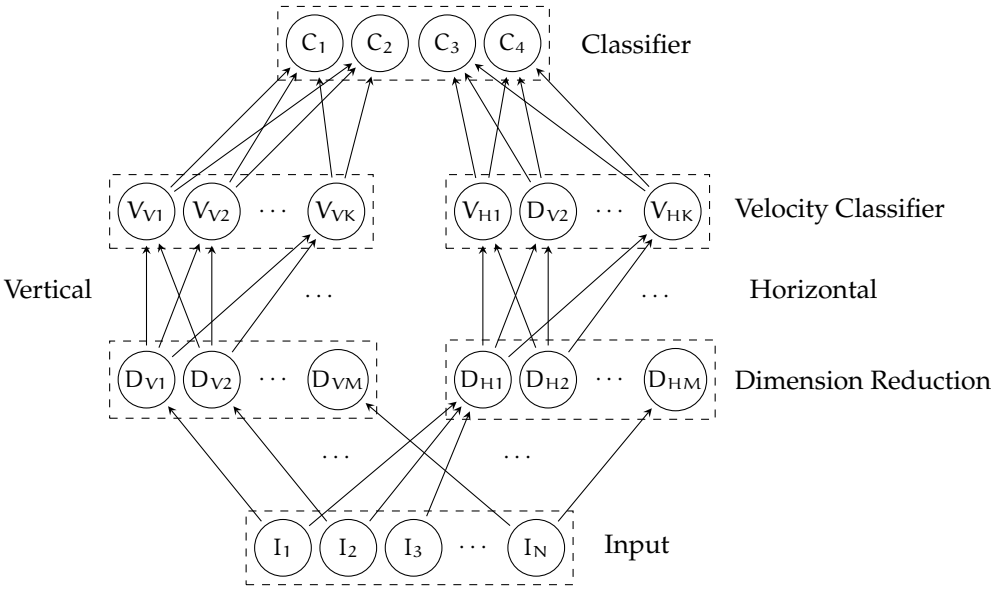

Figure 3: Topology used in the trained feed-forward approach.

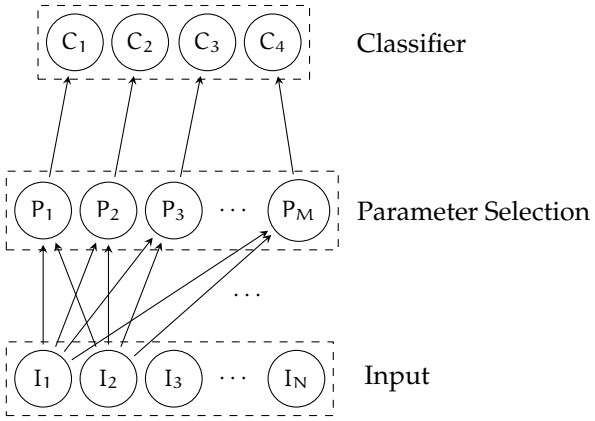

Figure 4: Topology used in the fixed feed-forward approach.

specific case. After testing different configurations, we retain three parameters to define these cases: the orientation of the trajectory (Θ), the orthogonal shift (C) and the velocity (V). The incoming synaptic delays and weights are defined by the following equations:

$$d_{x,y} = \|\overrightarrow{P_{x,y}C}\| \times \cos\left(\widehat{P_{x,y}\Theta}\right) \times V$$
$$w_{x,y} = \begin{cases} 1 & \text{when } \min(d(P_{x,y}, \Theta_{min}), d(P_{x,y}, \Theta_{max})) < D \\ 0 & \text{otherwise} \end{cases}$$

where x and y are the synapse coordinates on the input grid, $P_{x,y}$ the point at coordinates (x, y), C a reference point of the trajectory, V the current velocity, Θ the current orientation, Θ_{min} and Θ_{max} the orientation class bounds and d the perpendicular distance from a point to a line. The second layer aims to map each selection neuron to the associated class. According to the orientation parameter of each neuron of the first layer, a unique outgoing connection will be created to the classifier neuron associated to the orientation class (see Figure 4).
This approach provides excellent results (see Figure 2), but requires a large number of neurons and synapses to cover enough parameters.

4. Energy Consumption

In many types of applications, using SNNs can help to save a large amount of energy as compared to the same applications on classic Von Neumann architectures. However, since a comparison with such architectures is very difficult to realize (estimating the energy consumption of a program instruction is a difficult issue because of the numerous complex hardware and software mechanisms involved), we compare only the different SNN approaches to themselves. The estimation of the energy consumption of our models can be computed by the following equations:

$$E_{dynamic} = |\text{fire}| \times E_{fire} + |\text{spike}| \times E_{spike}$$
$$E_{static} = \Delta t \times \left(P_{neuron} \times |\text{neuron}| + P_{synapse} \times |\text{synapse}|\right)$$
$$E_{total} = E_{dynamic} + E_{static}$$

with $|\text{fire}|$ the number of neuron firing events, E_{fire} the energy consumed when a neuron fires, $|\text{spike}|$ the number of spikes passing through the synapses, E_{spike} the energy needed to transmit a spike through a synapse, Δt the duration of the measurement, $|\text{neuron}|$ the number of neurons in the network, P_{neuron} the power dissipated by one neuron, $|\text{synapse}|$ the number of synapses in the network, and P_{synapse} the power dissipated by one synapse.

Table 1: Estimation of the parameters of energy consumption for our hardware model.

Parameter	**Value**
E_{fire}	4 fJ
E_{spike}	4 fJ
P_{neuron}	100 pW
P_{synapse}	100 pW

Table 2: Estimation of the energy consumption of the different approaches. The results are obtained by averaging over 100 runs. Our estimations show that the consumed dynamic energy is negligible.

Network	$\mathbf{E}_{\text{dynamic}}$ (pJ)	$\mathbf{E}_{\text{static}}$ (µJ)	$\mathbf{E}_{\text{total}}$ (µJ)
RC $4 \cdot 4 \cdot 4$	261	8.5	8.5
RC $7 \cdot 7 \cdot 7$	1369	100.30	100.31
RC $10 \cdot 10 \cdot 10$	4003	526.24	526.25
Trained FF	227	2.62	2.62
Fixed FF	1303	102.04	102.04

Table 1 lists the parameters of our hardware model and Table 2 shows the simulated energy consumption of the three architectures considered. While conventional artificial neurons exhibit an energy efficiency in the range of 1 pJ/spike, it is worth noting that the dynamic power is far much lower than the static power in our case [10]: as described by table 2, the energy efficiency of the implemented model (4 fJ per spike) is negligible when compared to the static power (100 pW).

By comparing the energy consumption and the task performance of the three approaches, we see that they each have their own benefits. On the one hand, reservoir computing is the most general approach. However, in order to obtain satisfactory results, it is necessary to use a large reservoir, which rapidly increases the energy consumption. On the other hand, using a fixed network yields excellent results on the reference dataset. But again, a large network is required to have a good coverage of the parameters. Thus, the trained feed-forward network is a good candidate because it provides a very good compromise between the network size, and so the energy consumption, and the task performance. Even if this trained feed-forward network remains task-specific, it can be retrained on a different dataset.

5. Conclusion

We have demonstrated the flexibility of N2S3, our hardware spiking neural network simulator on a motion detection case study. This flexibility concerns the network topologies (feed-forward or recurrent), the learning approaches (local or global, supervised or unsupervised, weight or delay learning). We have also shown that we can evaluate both the generalization performance and the energy consumption of these various networks built with a very low power CMOS design.

Acknowledgement

This work has been partly funded by IRCICA (Univ. Lille, CNRS, USR 3380 – IRCICA, F-59000 Lille, France) as the Bioinspired project.

Bibliographie

1. Bichler (O.), Roclin (D.), Gamrat (C.) et Querlioz (D.). – Design exploration methodology for memristor-based spiking neuromorphic architectures with the Xnet event-driven simulator. – In *2013 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 7–12, juillet 2013.
2. Boulet (P.), Devienne (P.), Falez (P.), Polito (G.), Shahsavari (M.) et Tirilly (P.). – *N2S3, an Open-Source Scalable Spiking Neuromorphic Hardware Simulator*. – report, Université de Lille 1, Sciences et Technologies ; CRIStAL UMR 9189, janvier 2017.
3. Brette (R.), Rudolph (M.), Carnevale (T.), Hines (M.), Beeman (D.), Bower (J. M.), Diesmann (M.), Morrison (A.), Goodman (P. H.), Harris (F. C.), Zirpe (M.), Natschläger (T.), Pecevski (D.), Ermentrout (B.), Djurfeldt (M.), Lansner (A.), Rochel (O.), Vieville (T.), Muller (E.), Davison (A. P.), El Boustani (S.) et Destexhe (A.). – Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of Computational Neuroscience*, vol. 23, n3, décembre 2007, pp. 349–398.
4. Goodman (D. F. M.) et Brette (R.). – Brian: a simulator for spiking neural networks in Python. *Frontiers in Neuroinformatics*, vol. 2, 2008.
5. Izhikevich (E. M.). – Polychronization: computation with spikes. *Neural Computation*, vol. 18, n2, février 2006, pp. 245–282.
6. Jo (S. H.), Chang (T.), Ebong (I.), Bhadviya (B. B.), Mazumder (P.) et Lu (W.). – Nanoscale Memristor Device as Synapse in Neuromorphic Systems. *Nano Letters*, vol. 10, n4, 2010, pp. 1297–1301.
7. Merolla (P. A.), Arthur (J. V.), Alvarez-Icaza (R.), Cassidy (A. S.), Sawada (J.), Akopyan (F.), Jackson (B. L.), Imam (N.), Guo (C.), Nakamura (Y.), Brezzo (B.), Vo (I.), Esser (S. K.), Appuswamy (R.), Taba (B.), Amir (A.), Flickner (M. D.), Risk (W. P.), Manohar (R.) et Modha (D. S.). – A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, vol. 345, n6197, août 2014, pp. 668–673.
8. Paugam-Moisy (H.) et Bohte (S.). – Computing with Spiking Neuron Networks. *In : Handbook of Natural Computing*, éd. par Rozenberg (G.), Bäck (T.) et Kok (J. N.), pp. 335–376. – Springer Berlin Heidelberg, 2012.
9. Paugam-Moisy (H.), Martinez (R.) et Bengio (S.). – A supervised learning approach based on STDP and polychronization in spiking neuron networks. 2006.
10. Sourikopoulos (I.), Hedayat (S.), Loyez (C.), Danneville (F.), Hoel (V.), Mercier (E.) et Cappy (A.). – A 4-fJ/spike Morris-Lecar artificial neuron in 65nm CMOS technology. – In *5th Neuro Inspired Computational Elements Workshop (NICE)*, San Jose, United States, mars 2017.