Computational methods for event-based signals and applications

Xavier Lagorce

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité : Robotique

École doctorale : "Sciences mécaniques, acoustique, électronique & robotique de Paris"

réalisée à l'

Institut de la Vision

présentée par

Xavier Lagorce

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURRIE

Sujet de la thèse :

Computational Methods for Event-Based signals and Applications

Dirigée par:
Prof. Ryad Benosman

présentée et soutenue publiquement le 22 septembre 2015

Devant un Jury composé de :

Prof.	Elisabetta Chicca	Rapporteur
Prof.	Bernabé Linares-Barranco	Rapporteur
Prof.	Stéphane Viollet	Examinateur
Prof.	Stéphane Régnier	Examinateur
Associate Prof.	Sio-Hoi Ieng	Examinateur
Prof.	Ryad Benosman	Directeur de thèse

Institut de la Vision
Université Paris 6 / INSERM / CNRS
17 rue Moreau, 75012 PARIS

It is during my Ph.D. on *Computational Methods for Event-Based signals and Applications* at the Vision Institute, that I have discovered the novel and exciting world of Neuromorphic Engineering. Coming from a standard signal processing and computer vision background, this new way of observing the world and its signals was a radical, but very rewarding experience.

When I joined Prof. Ryad Benosman's team back in 2011, the work on these new asynchronous, event-based sensors had just begun and almost everything was to be done. Since then, the team grew and took an important place in the field of event-based processing. The very exploratory work linked to this kind of adventure led me to play a role in a large number of very different projects which made no day look like another one.

These four years of Ph. D. were a formidable adventure which led to unforeseen territories and lots of uncharted lands. I also met several great collaborators during these years, some of them I am now proud to consider as friends.

I will not enumerate everybody who had an important impact during this period by fear of forgetting someone, but let them be thanked for all they did. I will nevertheless emphasize a few of them:

- *Mr. R.* for all his remarks on *not good enough*-sentences and figures and all these future-thingy we designed or thought of together,
- *Michel* for his advice about the aforementioned remarks,
- *Christoph* for his *perfect* sensor,
- Everybody from the *Vision Institute* and it's *Vision and Natural Computation* team for their input and all the good memories I will keep from spending time, working and having coffee with them,
- The paintball team for the pain I still feel when the weather is bad,
- *Rancilio Silvia* for real unending tweaking and experimentation,
- and *The Game* for hours of fun.

Abstract

Computational Methods for Event-Based signals and Applications: abstract

Computational Neurosciences are a great source of inspiration for data processing and computation. Nowadays, how great the state of the art of computer vision might be, it is still way less performant that what our brains or the ones from other animals or insects are capable of. This thesis takes on this observation to develop new computational methods for computer vision and generic computation relying on data produced by event-based sensors such as the so called "silicon retinas".

These sensors mimic biology and are used in this work because of the sparseness of their data and their precise timing: information is coded into *events* which are generated with a microsecond precision. This opens doors to a whole new paradigm for machine vision, relying on time instead of using images. We use these sensors to develop applications such as object tracking or recognition and feature extraction. We also used computational neuromorphic platforms to better implement these algorithms which led us to rethink the idea of computation itself.

This work proposes new ways of thinking computer vision via event-based sensors and a new paradigm for computation. Time is replacing memory to allow for completely local operations, enabling highly parallel machines in a non-Von Neumann architecture.

Méthodes de Calcul pour les Signaux Événementiels et Applications : résumé

Les neurosciences computationnelles sont une grande source d'inspiration pour le traitement de données. De nos jours, aussi bon que soit l'état de l'art de la vision par ordinateur, il reste moins performant que les possibilités offertes par nos cerveaux ou ceux d'autres animaux ou insectes. Cette thèse se base sur cette observation afin de développer de nouvelles méthodes de calcul pour la vision par ordinateur ainsi que pour le calcul de manière générale reposant sur les données issues de capteurs événementiels tels que les "rétines artificielles".

Ces capteurs copient la biologie et sont utilisés dans ces travaux pour le caractère épars de leurs données ainsi que pour leur précision temporelle : l'information est codée dans des *événements* qui sont générés avec une précision de l'ordre de la microseconde. Ce concept ouvre les portes d'un paradigme complètement nouveau pour la vision par ordinateur, reposant sur le temps plutôt que sur des images. Ces capteurs ont été utilisés pour développer des applications comme le suivi ou la reconnaissance d'objets ou encore de l'extraction de motifs élémentaires. Des plateformes de calcul neuromorphiques ont aussi été utilisées pour implémenter plus efficacement ces algorithmes, nous conduisant à repenser l'idée même du calcul.

Les travaux présentés dans cette thèse proposent une nouvelle façon de penser la vision par ordinateur via des capteurs événementiels ainsi qu'un nouveau paradigme pour le calcul. Le temps remplace la mémoire permettant ainsi des opérations complètement locales, ce qui permet de réaliser des machines hautement parallèles avec une architecture non-Von Neumann.

Contents

List of Figures

Introduction

SINCE the late 1980s with the work of Carver Mead [1, 2, 3], the field of Neuromorphic Engineering has been a growing field focusing on several applications [4, 5, 6, 7, 8], from auditory processing to vision and processing chips. This Ph.D. work is however mainly focused on biomimetic event-driven time-based vision sensors such as the ones developped in [9, 10].

Event Based cameras - like the biological retina - are driven by *events* happening within the visual scene instead of artificially created timings like conventional vision sensors. When my Ph.D. work started, we were using the DVS silicon retina [11] to begin exploring this frameless acquisition principle. We started rapidly rethinking and translating standard computer vision algorithms from the *conventional* concept of frames to a pure time based framework.

During this work, it became clear that a deep understanding of the sensor was necessary to develop robust processing event based techniques. We were then joined by chip designers who had been developing their own silicon retina: the ATIS [12, 13]. My involvement in the prototyping of the different versions of the ATIS PCB enabled me to know more about the chip and brought me useful insights in its principle of operations. This also fueled the idea of developing a multithreaded software architecture to develop algorithms for these cameras. This led to the development of kAER, a SDK able to manage event based processing, easy prototyping and real-time implementation of all the algorithms developed in the team. During my Ph.D., I continued updating and improving the software, until its recent acquisition by the Lab's spinoff Chronocam.

The beginning of my Ph.D. was focused on tracking moving object and defining some new kind of features adapted to event-based signals.

In Chapter 1, we focus on object tracking and introduce two methods using the dynamic content of a stream of events produced by an asynchronous camera such as the DVS or the ATIS. The change detection properties of the sensor natively allow these techniques to be very robust to different lighting conditions thus allowing unprecedented performances.

The first technique introduced in this work is to approximate cluster of events by mathematical objects such as Gaussian shapes. It enables to simultaneously track the position and the shape of an object in the focal plane.

The second technique extends the method to any arbitrary kernel. The process only needs few requirements to operate. The trackers are then not only tracking objects but are also detecting shapes in the field of view of the sensor. This detection is made possible by the organization of the trackers in a pool. Prototypes of trackers are covering the whole focal plane, ready to track an object that would match their kernels. If such an object is detected, the tracker gets activated and

starts following this particular object or feature in the scene until it disappears and the tracker is destroyed. Very low computational power is needed by each tracker, this then allows to compute in real-time a larger number of such trackers. Several pools of trackers can be used at the same time to either track different objects or different scales of the same object.

In the work of Chapters 2 and 3, we focused on feature extraction. Similarly to existing computer vision concepts, common feature extraction techniques are adapted to frames, and more than not being directly applicable on events, they completely ignore the dynamics of a scene. Exploiting both spatial and temporal information contained in the output of neuromorphic sensors was then a key point of the features we have developped.

In Chapter 2, we use an unsupervised neural network based on a set of Echo-State Networks (ESN). These recurrent neural networks are capable of extracting dynamics from their input. The proposed architecture consists of a number of these networks where events are sent to their inputs. Each ESN goal is to predict the evolution of its input. ESN also compete, only the best predictor is allowed to learn the presented input at a given time. This selection process results in the specialization of each network in the prediction of particular patterns. One can then deduce the presence of a particular pattern, or feature, in the input scene when a given network is selected. Because of the recurrent connections of the ESNs, the extracted feature is not only spatial but also temporal.

In Chapter 3, we pushed the idea further by directly including this temporal information in a representation of the events. Each input spike from the sensor is associated to a *context* describing the recent history of its neighboring pixels. All contexts can be clustered in an unsupervised manner around a given number of centers. These centers can then be seen as a set of spatio-temporal features describing the scene that the camera is looking at. From these, it is possible to build a hierarchical architecture relying on several layers of such context building and clustering. This increases the complexity of the contexts by increasing their timescale and spatial size. At the end, the feature activation of the last layer of this architecture contains a high-level spatio-temporal information about the presented stimuli.

The proposed algorithms are very promising and well adapted to the event-based representation of the visual information produced by neuromorphic cameras. However, they are not well suited for computation on standard computers. Their local properties and asynchronous nature allow them to be natural candidates for highly-parallel platforms. Standard off-the-shelf computers are mostly linear pipelines processing all the input events sequentially. This lack of adequation between our algorithms and the platforms used to implement them motivated us to start looking in other directions. The first step was to find new computing platforms, highly-parallel and as asynchronous as possible. A good candidate is the SpiNNaker platform [14]. This platform is consituted of numerous small processors (the final version aims at embedding 1 million processors) linked together on a fabric designed for high-speed communication of small packet, from one end of the system to another. To enable fast prototyping with this platform, the SpiNNaker team provides a SDK allowing to implement neural networks and to configure a SpiNNaker machine. Unfortunately, the available SDK did not cover all our needs, we therefore started exploring and extending the SDK ourselves. This is the main focus of the 2 following chapters.

In Chapter 4, we worked on an improvement of the implementation of plasticity implementation on SpiNNaker, learning having such a central part in several of our algorithms. Instead of mixing the code simulating neural populations and the code managing synaptic weights in the network, we separated the two in different processors. This allows to optimize the memory accesses

needed to route spikes between cores for simulating the network itself and the ones needed to update the weights and connections. The capabilities of this method were demonstrated on several examples of the literature and results show a large improvement over the standard implementation of plasticity provided with the official SpiNNaker software.

In Chapter 5, we tackled the problem of time resolution. When designing the official SpiNNaker software, the choice was made to simulate all the neural networks with a millisecond time precision as a trade-off between several constraints of the machine. This is enough for lots of applications but very limiting for others such as auditory applications. Sound localization for instance, requires a time precision in the order of the hundred of microseconds or less. After characterizing the time constraints of the communication fabric of the SpiNNaker machines, we proposed a new implementation of the simulation infrastructure to obtain fully asynchronous models which can then cope with a microsecond time resolution. To continue on what was achieved in the previous chapter, plasticity was also reimplemented usng these time scales.

Being able to use such a highly-parallel platform as SpiNNaker brought us a very interesting insights into the problems encountered with such systems. It was clear, we had to go further and take another step forward. Taking standard processors and linking them together, even as smartly as the SpiNNaker team does is not enough. The machine itself has to be thought again.

No more memory, only time.

This is the focus of Chapter 6. Using a time-based representation of data and neuron-like units (using the same principles and basic rules as biological neurons), we derive neural-like circuits able to compute using data expressed as time intervals. This lays the development of a non von Neumann architecture of a neuromorphic computer. The framework uses spiking neurons to store data, compute any linear or non linear operators and relational tools. From there any algorithm can be implemented easily. The computing system is parallel, neuromorphic and it produces data only when necessary. This framework offers a way to systematically *program* large neural circuits.

All of the chapters presented in this document have led to publications [15, 16, 17, 18] as well as many rewarding collaborations [19, 20, 21, 22, 23]. A full list of publications, including some not yet published papers can be found in Appendix A as well as a list of patents related to the work I have done during my Ph.D. in Appendix B.

Chapter 1

Asynchronous Event-Based Multi-kernel Algorithm for High Speed Visual Features Tracking

This chapter presents a number of new methods for visual tracking using the output of an event-based asynchronous neuromorphic dynamic vision sensor. It allows the tracking of multiple visual features in real-time, achieving an update rate of several hundred kilohertz on a standard desktop PC. The approach has been specially adapted to take advantage of the event-driven properties of these sensors by combining both spatial and temporal correlations of events in an asynchronous iterative framework. Various kernels, such as Gaussian, Gabor, combinations of Gabor functions and arbitrary user-defined kernels are used to track features from incoming events. The trackers described in this work are capable of handling variations in position, scale and orientation through the use of multiple pools of trackers. The tracking performance was evaluated experimentally for each type of kernel in order to demonstrate the robustness of the proposed solution.

Contents

1.1 Introduction

Visual object recognition and tracking is useful in many applications, such as video surveillance, traffic monitoring, motion analysis, augmented reality and autonomous robotics. Most object tracking techniques rely on sequences of static frames which limits algorithmic efficiency when dealing with highly dynamic scenes. Conventional frame-based video cameras can acquire data at frequency as high as several tens of kilohertz, nevertheless this amount of data remains difficult to process in real-time due to the large amount of redundant acquired information. Real-time processing at high acquisition rates usually requires different techniques such as: sub-sampling of the field of view [24], the use of specific hardware implementations or a restriction to simple tracking algorithm such as image's centroid and moments computation [25].

This chapter presents an event-based approach to fast visual tracking of features using the output of an asynchronous neuromorphic event-based camera. Neuromorphic cameras (sometimes called silicon retinas) mimic the biological visual systems [10][9]. The Asynchronous Time-based Image Sensor (ATIS) camera used in this work reacts to changes of scene contrast and records only dynamic information, thus reducing the data quantity [13][12]. The presented algorithms rely on the change detection circuit of the ATIS, namely it only deals with relative change events. It can also be applied to other sensors such as the Dynamic Vision Sensor (DVS) [9] on which the ATIS is based.

The presented tracking algorithm is computationally inexpensive and is thus capable of tracking objects and updating their properties at rates in the order of hundreds of kilohertz. Firstly, an asynchronous event-based Gaussian blob tracking algorithm is developed and examined. The properties of the Gaussian kernel allow adaptation to the events' spatial distribution by continuously correcting the Gaussian size, orientation and location with each incoming event. This provides the object's position, size, and orientation simultaneously. In a second stage, the model is extended by using oriented Gabor kernels that allow tracking specific oriented edges. This work also considers the combination of several oriented kernels that are useful in tracking specific focal plane structures such as corners. Finally, a general kernel approach is presented. It can use almost any arbitrary kernel and can be seen as a generalization of the process, with the only constraint that their center of mass has to be aligned with the center of the kernels (see section 1.3.3.4).

1.2 Time encoded imaging

Biomimetic event-driven time-based vision sensors are a novel class of vision device that - like the biological retina - are driven by "events" happening within the visual scene. They are not like conventional vision sensors, which are driven by artificially created timing and control signals (e.g. frame clock) that have no relation whatsoever to the source of the visual information [26]. Over the past few years, a variety of these event-based devices has been developed, including temporal contrast vision sensors that are sensitive to relative luminance change [13, 27, 26], gradient-based sensors sensitive to static edges [28], and optical-flow sensors [29]. Most of these vision sensors output visual information about the scene in the form of asynchronous address events using the Address Event Representation (AER) protocol [30] and encode the visual information in the time dimension rather than as a voltage, charge, or current. The presented pattern tracking method is designed to work on the data delivered by such a time-encoding sensor and takes full advantage of the high temporal resolution and the sparse data representation.

The ATIS used in this work is a time-domain encoding vision sensors with 304x240 pixels resolution. [12]. The sensor contains an array of fully autonomous pixels that combine an illuminance relative change detector circuit and a conditional exposure measurement block.

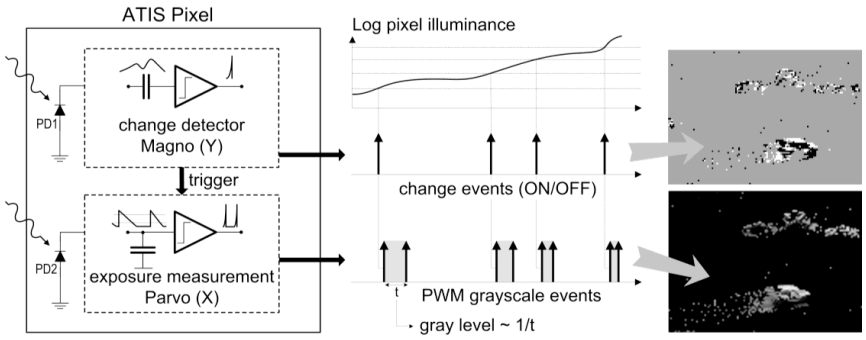


Figure 1.1: Functional diagram of an ATIS pixel [13]. Two types of asynchronous events, encoding change and brightness information, are generated and transmitted individually by each pixel in the imaging array.

As shown in the functional diagram of the ATIS pixel in Fig. 1.1, the relative change detector individually and asynchronously initiates the measurement of an exposure/gray scale value only if - and immediately after - a brightness change of a certain magnitude has been detected in the field-of-view of the respective pixel. The exposure measurement circuit in each pixel individually encodes the absolute instantaneous pixel illuminance into the timing of asynchronous event pulses, represented as inter-event intervals.

Since the ATIS is not clocked like a conventional camera, the timing of events can be conveyed with a very accurate temporal resolution in the order of microseconds. The time-domain encoding of the intensity information automatically optimizes the exposure time separately for each pixel instead of imposing a fixed integration time for the entire array, resulting in an exceptionally high dynamic range and improved signal to noise ratio. The pixel-wise change detector driven operation yields almost ideal temporal redundancy suppression, resulting in a maximally sparse encoding of the image data.

In what follows we will rely only on change detector events as the timings of events is the main needed information to perform tracking.

1.3 Multi kernel event-based features tracking

This section describes the tracking algorithms developped in this work. After reviewing the state of the art of asynchronous tracking using event-base silicon retinas, we describe how bivariate normal distributions are used to track clouds of events. Then, we will generalize our approach to more arbitrary kernels. The next two sub-sections tackle the problem of tracking several objects at once in a scene and how several trackers interact with one another in a pool mechanism. To conclude this section, a global algorithm is presented and a number of possible optimizations are proposed.

1.3.1 State of the Art

Image segmentation, feature extraction, optical flow and high level motion filters are usually used to track moving objects but they are known to be computationally expensive tasks. Real-time processing with frame rates reaching several hundreds of hertz can lead to more robust tracking algorithms [31]. However, it is currently an almost impossible task to perform such tasks in real-time unless dedicated hardware is used. Dedicated hardware solutions introduce additional implementation complexity that limits the efficiency of such vision algorithms. Processing at such a high frame rate is only applicable for tackling simple tasks such as the detection of the center of mass (or centroid) of moving objects [32, 33].

The newly developed event-based silicon retinas (Dynamic Vision Sensor [26], Asynchronous Time-based Image Sensor [13]) inspired by the physiology of biological retinas are promising sensors for fast vision applications. These sensors convey a sparse stream of asynchronous time-stamped events suitable for object tracking as only dynamic information is captured. Several tracking algorithms have been developed for this type of sensor. An event clustering algorithm is introduced for traffic monitoring, where clusters can change in size but are restricted to a circular form [34, 35]. A fast sensory motor system has been built to demonstrate the sensor's high temporal resolution properties in [36]. Several event-based algorithms and a remarkable JAVA framework for the Dynamic Vision Sensor can be found at [37]. In [38], a pencil balancing robot is developed to stabilize a pencil using a fast event-based Hough transform. The sensor has been recently applied to track particles in microrobotics [39] and in fluid mechanics [40]. It was also used to track a micro-gripper's jaws to provide a real-time haptic feedback from the micro meter scale world (lengths and sizes of objects around 1e-6m) [41].

To date, the currently developed methods that operate on events focus on the extraction of features such as lines in the whole focal plane. In other cases, they are too general to deal with particular cases where trackers should locally follow a specific oriented edge or a local shape. We will thus extend this methods to provide a more general framework allowing the tracking of specific local features using an event-based methodology. The tracking approach proposed here is inspired by the mean-shift technique that has also been extensively used in conventional frame-based visual tracking [42, 43, 44].

1.3.2 Gaussian Blob tracking

A stream of visual events can be mathematically defined as follows: let $ev(\mathbf{u},t) = [\mathbf{u},t,pol]^T$ be a quadruplet giving the pixel position $\mathbf{u} = [x,y]^T$, the time t of the event and pol, its polarity that can be -1 or 1. When an object moves, the pixels generates events which geometrically form a point cloud that represents the spatial distribution of the observed shape. A moving object generates events that follow a spatial distribution that can be, in a first stage, roughly approximated by a bivariate normal distribution $\mathcal{N}(\mu,\Sigma)$, also called bivariate Gaussian distribution). The parameters of $\mathcal{N}(\mu,\Sigma)$ provide the object's position, size and orientation. The Gaussian mean μ indicates the object's position while the covariance matrix Σ represents its size and orientation. Let us suppose that several Gaussian trackers have already been initialized on the focal plane's locations of several moving objects. When a new event occurs, it is assigned a score (up to the normalization term) for each tracker, inspired by the probability this event would be generated by the Gaussian distribution

associated to this tracker which can be calculated by :

$$p_i(\mathbf{u}) = \frac{1}{2\pi}|\Sigma_i|^{-\frac{1}{2}} e^{-\frac{1}{2}(\mathbf{u}-\mu_i)^T\Sigma^{-1}(\mathbf{u}-\mu_i)} \tag{1.1}$$

where $\mathbf{u} = [x,y]^T$ is the pixel location of the event. $\mu_i = [\mu_{ix},\mu_{iy}]^T$ represents the i^{th} tracker's location and $\Sigma_i \in \mathbb{R}^{2\times 2}$ is its covariance matrix:

$$\Sigma = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_y^2 \end{bmatrix} \tag{1.2}$$

Fig. 1.2 is showing an example of a ellipse shape tracked by such a Gaussian tracker.

If one of the computed scores is superior to a predefined threshold $p_i(\mathbf{u}) > \delta p$ (usually set to 0.1), then this tracker will adapt its parameters to follow this incoming event. If several trackers respond to the same event, the one with the maximum score is always chosen. After the most probable tracker has been selected, its parameters can be updated using a simple weighting strategy by integrating the last distribution with the current event information (see Eq (1.3) and (1.4)). Since only the chosen tracker is examined hereafter, the subscript i indicating the tracker number is omitted for clarity. The position and size of a Gaussian tracker can be updated as follows:

$$\mu_t = \alpha_1\mu_{t-1} + (1-\alpha_1)\mathbf{u} \tag{1.3}$$

$$\Sigma_t = \alpha_2\Sigma_{t-1} + (1-\alpha_2)\Delta\Sigma \tag{1.4}$$

where α_1 and α_2 are the update factors. They should be adjusted according to the event rate and the nature of the observed scene. These values are typically set between 10^{-2} and 10^{-1} and are chosen according to the size and velocity of the tracked objects. Namely how many events should be received to drag the shape from its old position to the new one and to change its size and shape according to the incoming events' rates.
The covariance difference $\Delta\Sigma$ is computed using the current tracker's location $\mu_\mathbf{t} = [\mu_{tx},\mu_{ty}]^T$ and event's location $\mathbf{u}$:

$$\Delta\Sigma = \begin{bmatrix} (x-\mu_{tx})^2 & (x-\mu_{tx})(y-\mu_{ty}) \\ (x-\mu_{tx})(y-\mu_{ty}) & (y-\mu_{ty})^2 \end{bmatrix}. \tag{1.5}$$

Finally, we define the activity of each tracker $\mathcal{A}_i$ that is updated at each incoming event $ev(\mathbf{u},t)$, following an exponential decay function which describes the temporal dimension of the Gaussian kernel.

$$\mathcal{A}_i(t) = \begin{cases} \mathcal{A}_i(t-\Delta t)e^{-\frac{\Delta t}{\tau}} + \quad p_i(\mathbf{u}), \\ \qquad\qquad \text{if } ev(\mathbf{u},t) \text{ belongs to tracker } i \\ \mathcal{A}_i(t-\Delta t)e^{-\frac{\Delta t}{\tau}}, \\ \qquad\qquad \text{otherwise.} \end{cases} \tag{1.6}$$

where Δt is the time difference between current and previous events and τ tunes the temporal activity decrease. This activity measure is useful for inhibition and repulsion procedures that will be explained latter. It allows us to shape the interaction between trackers.

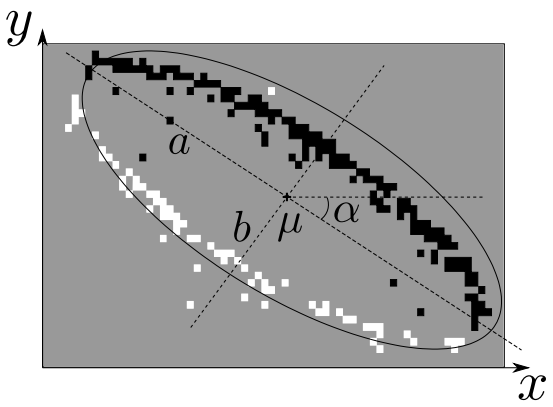


Figure 1.2: Error ellipse of the covariance matrix of a Gaussian tracker following a black ellipse moving under a white background. Position, size and orientation of the tracker automatically adapt to the visual stimuli. Black and white dots represent respectively OFF and ON events.

Remark

The object's size and orientation can be retrieved by computing the principal components of the covariance matrix. Computationally, the lengths of the error ellipse's axes are the two principal components of the covariance matrix, which can be explicitly calculated by decomposing the two eigenvalues λ_{min} and λ_{max}. The semi-axes a and b and the orientation angle α of the ellipse can be computed as follows:

$$a = K\sqrt{\lambda_{max}} \tag{1.7}$$

$$b = K\sqrt{\lambda_{min}} \tag{1.8}$$

$$\alpha = \frac{1}{2} arctan\left(\frac{2\sigma_{xy}}{\sigma_y^2 - \sigma_x^2}\right) \tag{1.9}$$

where K is a scaling factor describing the distribution's confidence level. K is given by the confidence intervals, it provides a tuning parameter linking the Gaussian distribution parameters to the size of the event cloud it is fitted to in the image plane. In practice, this value can be set to 1. Readers wishing to know more about the computation of confidence intervals should refer to [45].

Computing these parameters can provide additional information if the trackers are used to identify unknown objects in a scene. The size and orientation of the Gaussian distributions yield information about the shape or orientation of the tracked objects and can be used either to discriminate between interesting and irrelevant objects or to build more complex objects from several trackers.

1.3.3 Multi-kernel features tracking

1.3.3.1 Principle

the principle is very similar to the Gaussian tracker, except that the Gaussian kernel is replaced by various other kernels. The examples that will be illustrated here are Gabor oriented kernels, combinations of Gabor functions, Laplacian of Gaussian and more general handmade kernels such as a triangle and a square. Some of these kernels are shown in Fig. 1.3.

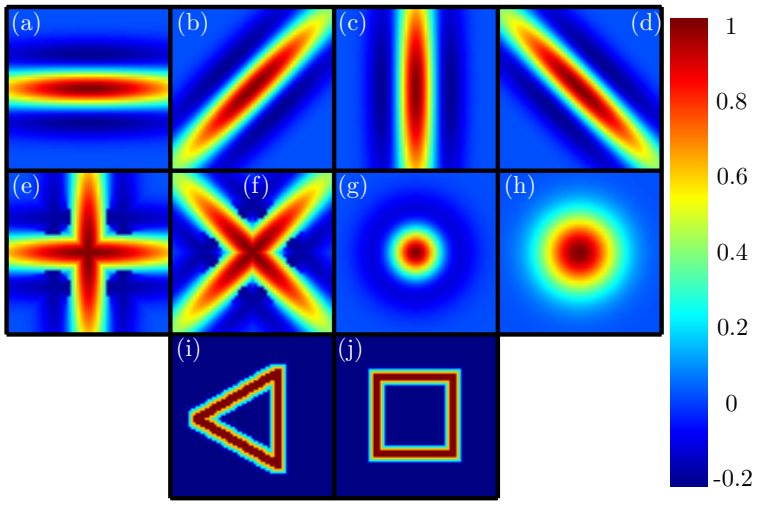


Figure 1.3: Examples of kernels used for tracking. (a) to (d) Gabor oriented kernels. (e) and (f) Combinations of Gabor kernels. (g) Laplacian of Gaussian. (h) Gaussian kernel. (i) and (j) Handmade arbitrary kernels.

The feature tracker generalizes to almost any kernel even for kernels with no analytical form. If K_i has no closed form, then it is a numericaly defined function. In this case, the update of size and orientation cannot be performed easily as this implies the estimation of rotation and scaling that go beyond the scope of this work. We chose, in this case, to restrict kernels to a fixed size (scale) and orientation. Due to the low computational cost, this can easily be supplemented by the use of multiple layers of kernels of different scales and orientations as will be shown in section 1.4.

We introduce a local inhibition procedure that prevents the tracker from being active if the same localized area of the kernel is always excited. The activity of tracker i, is then given by Eq (1.10).

$$\mathcal{A}_i(t) = \begin{cases} \mathcal{A}_i(t-\Delta t)e^{-\frac{\Delta t}{\tau}} + \\ \qquad G_{\text{inhib},i}(ev,\mathbf{u})K_i(x_i-x,y_i-y), \\ \text{if } ev(\mathbf{u},t) \text{ occurs into receptive field of tracker } i \\ \mathcal{A}_i(t-\Delta t)e^{-\frac{\Delta t}{\tau}}, \text{otherwise,} \end{cases} \tag{1.10}$$

where $\mathbf{u} = [x_i, y_i]^T$, is the position of tracker i, K_i is the matrix representing the tracker's kernel and G_{inhib}, the inhibition gain, is computed as follows :

$$G_{\text{inhib},i}(ev,\mathbf{u}) = 1 - e^{-\Delta t_{ev,\mathbf{u}}/\tau_{inhib}} \tag{1.11}$$

Local inhibition is ensured considering $\Delta t_{ev,\mathbf{u}}$ as the temporal difference between event ev and the last occurrence of an event into a small spatial neighborhood of the point $[x_i - x, y_i - y]^T$ in matrix K_i (we usually considered a neighborhood of one or two pixels radius). This mechanism is equivalent to a local inhibition of neighboring locations of matrix K_i where the event occurred.

Namely the activated location of the filters and its surrounding will no longer be able to drive the tracker even if an event reactivates that location. In that case, $\Delta t_{ev,\mathbf{u}}$ will go towards 0 thus leading G_{inhib} to 0. As the temporal distance increases, G_{inhib} will gradually tend towards 1. This ensures that a kernel needs to receive events in all its area to lead to the tracker's activation. This inhibition is followed by an exponential reactivation of the inhibited area thus allowing the filter to be active to events arriving at that location.

1.3.3.2 Gabor oriented kernels and combinations of Gabors

it is well known that some areas of our visual cortex V1 respond preferentially to oriented stimuli [46]. Consequently, we used Gabor functions to build a model of orientation selective kernels. The response of a θ-oriented Gabor kernel $\mathcal{K}_{G\theta}(\mathbf{v},\sigma)$ located at position $\mathbf{v} = [x_G, y_G]^T$ to the incoming event $ev(\mathbf{u},t)$ is given by Eq (1.12):

$$G(e, \mathcal{K}_{G\theta}(\mathbf{v},\sigma)) = e^{\left(-\frac{x_{\mathbf{u},\mathbf{v}}^2+\gamma^2 y_{\mathbf{u},\mathbf{v}}^2}{2\sigma^2}\right)} \cos\left(2\pi\frac{x_{\mathbf{u},\mathbf{v}}}{\lambda}\right), \tag{1.12}$$

where $x_{\mathbf{u},\mathbf{v}} = (x-x_G)\cos\theta + (y-y_G)\sin\theta$ and $y_{\mathbf{u},\mathbf{v}} = -(x-x_G)\sin\theta + (y-y_G)\cos\theta$.
To ensure a correct orientation selectivity, we set the parameters $\gamma = \frac{\sigma}{15}$ and $\lambda = 4\sigma$. Gabor kernels are illustrated in Fig. 1.3 (a) to (d). To track a particular feature like a cross, we also built kernels by combining Gabor functions following orthogonal orientations, as shown Fig. 1.3 (e) and (f).

1.3.3.3 Laplacian of Gaussian kernel

this kernel is inspired by center-surround biological structures and can be represented by the *Laplacian of Gaussian* function (LoG), shown in Fig. 1.3(g). This kernel has a behavior similar to Gaussian kernels shown in Fig. 1.3(h) but is also sensitive to the size of the tracked events' cloud. A cloud whose size exceeds the bright central ring (see Fig. 1.3(g)) will induce negative contributions to the activity of events occurring in the dark ring area. This compensates for the effect of centered events, preventing the tracker from being activated.

1.3.3.4 General kernels

in fact the algorithm can use any kernel, with the only constraint being that the center of mass of the matrix that represents the kernel has to be aligned with the center of the kernel. This is a weak constraint since any general kernel can be spatially shifted to meet this restriction arising from the position's update principle. Since the tracker is attracted by the neighboring events, its position will naturally match the local events' center of mass. To test if the events' cloud matches the desired feature, both must share the same spatial locations.
As an example, general kernels are shown using a square and a triangle (see Fig. 1.3 (i) and (j)). These kernels are generated using a binary representation of a simple geometric shape (+1 on lines, -1 elsewhere) followed by a dilatation algorithm and a smoothing.

1.3.4 Multi-target tracking

The algorithm is first initialized with a hidden layer of pre-constructed trackers that are uniformly distributed among the whole field of view. Hidden trackers refer to those not displayed on the screen as active trackers. A hidden tracker is one that does not represent a real object but serves to

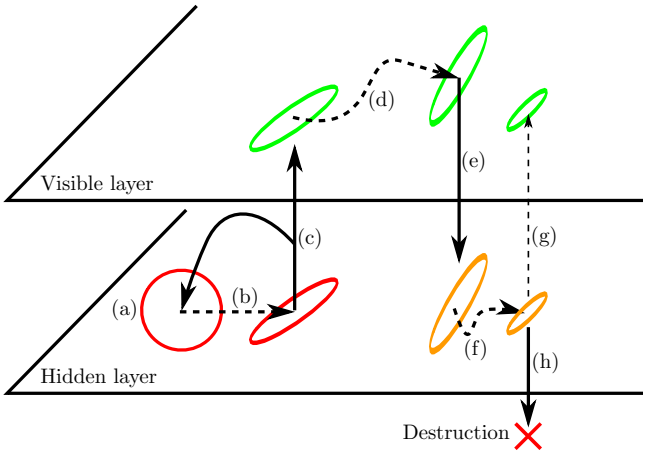


Figure 1.4: Activation scheme of trackers (Gaussian tracker in the example). (a) A tracker is initialized in the hidden layer with default parameters. (b) It starts to follow the event cloud from a moving object and adapts to its shape. (c) When its activity goes above $\mathcal{A}_{up}$ the tracker is upgraded to the visible layer and a new independent tracker is created in the hidden layer with its initial position and parameters. (d) The tracker follows the event cloud. (e) When its activity decreases and falls under $\mathcal{A}_{up}$, the tracker is downgraded to the hidden layer but keeps its current parameters and position. (f) Another step of tracking. (g) Eventually, the activity may rise again over $\mathcal{A}_{up}$ and the tracker can be upgraded again. (h) Finally, if its activity falls under $\mathcal{A}_{down}$, the tracker is deleted.

seed a potential tracker.
A tracker will be automatically attracted to the nearest events' cloud. When a new event occurs, and no active tracker responds to it, it directly feeds the nearest hidden tracker with the same update rule as described by equation (1.3) (see Fig. 1.4(b)). When the activity of a hidden tracker increases above a predefined threshold $\mathcal{A}_{up}$ (Fig. 1.4(c)), the tracker is upgraded to the visible layer, and a new hidden tracker is added with default parameters so that the hidden layer always has a fixed number of trackers distributed across the scene. When its activity falls under $\mathcal{A}_{down}$, the visible tracker is deleted (Fig. 1.4(h)). In the intermediary step, between $\mathcal{A}_{up}$ and $\mathcal{A}_{down}$ (Fig. 1.4(f)), the tracker is downgraded to the hidden layer but keeps its actual parameters and position. The evolution criteria combines both spatial and temporal correlation of events. Parameters $\mathcal{A}_{down}$, $\mathcal{A}_{up}$ and the temporal decrease constant τ have to be tuned according to the event rate and the desired behavior.

1.3.5 Mutual repulsion and attraction to initial position

In case of strong localized activity, all neighboring trackers can be attracted to the same location. In order to prevent the different trackers following the same cloud, a mutual repulsion process is added. Each time a tracker is activated by an event, the distance between each tracker pair is computed and the trackers that are too close are shifted away from each other, following the weighted repulsion function described in equation (1.13). The location μ_i of a tracker is updated

compared to the location of another tracker at μ_j as follows:

$$\mu_\mathbf{i} \leftarrow \mu_\mathbf{i} - \alpha_{rep} e^{-\frac{\|\mu_\mathbf{i}-\mu_\mathbf{j}\|}{d_{rep}}} \frac{\mathcal{A}_j^2}{\mathcal{A}_i^2+\mathcal{A}_j^2}(\mu_\mathbf{j}-\mu_\mathbf{i}), \tag{1.13}$$

where parameters α_{rep} and d_{rep} set the repulsion behavior. The mutual repulsion between trackers i and j is designed to ensure a correct repartition of trackers based on the distance that separates them but also on the activity of both trackers. The more a tracker is active (i.e. is efficiently tracking an event cloud), the less it will be influenced by its neighbors.
Due to this mutual repulsion, inactive trackers are often pushed far from their initial position by moving active trackers. Thus empty areas could grow on the field of view. To compensate this possibility, we also added an attraction force that impacts on each inactive tracker in the following way:

$$\mu_\mathbf{i} \leftarrow \begin{cases} \mu_\mathbf{i}^\mathbf{0}, & \text{if } \|\mu_\mathbf{i}-\mu_\mathbf{i}^\mathbf{0}\| > d_{max} \\ \mu_\mathbf{i} + \alpha_{att}(\mu_\mathbf{i}^\mathbf{0}-\mu_\mathbf{i}), & \text{otherwise.} \end{cases} \tag{1.14}$$

where $\mu_\mathbf{i}^\mathbf{0}$ is the initial position of tracker i. Thus, an inactive tracker that moved too far away from its initial position without being activated will be reset to its initial position.

Two additional constraints are added. Trackers that are near the border and about to move out of the focal plane are deleted. The borders are one pixel thick and a tracker is deleted if the distance of the its center to the border is below its typical size. For a Gaussian blob this typical size is the length of the ellipse's smallest axis (axis b in Fig. 1.2). For a symetric tracker (e.g. square, triangle) the size is the circumscribed circle's radius. The second constraint prevents the Σ matrix going below a low threshold value for the Gaussian trackers to limit the tracking to objects with a reasonable size. These additional constraints do not have much impact on the overall tracking performance as the sensor is usually configured to capture the scene in center of the focal plane.

1.3.6 Global algorithm

To summarize, the whole process of event-based features tracking is given by the following algorithm 1.

1.3.6.1 Remarks

For simplicity, repulsion and attraction are computed every time a new event occurs, but as trackers do not move that fast, it is possible to perform computation less frequently. When adding this optimization, processing the repulsion and attraction step can usually be done every millisecond and still yield good performances.
Events are considered regardless of their polarity as we are interested in the global position and orientation of the object. In what follows we will focus on a single polarity. This will be carried out to provide a precise measure of accuracy of the positionning of a tracker for a thin contour.

1.4 Results

The experiments have been carried out using the ATIS camera considering only its change detection output and gray levels have not been used. All programs have been written in C++ under

Algorithm 1 Event-based features tracking Algorithm

for every incoming event $ev(\mathbf{u},t)$ **do**
  Update the activity $\mathcal{A}_i$ of each tracker of the visible layer using Eq (1.6) or (1.10).
  Update the best candidate tracker's position (and size) using equations (1.3) (and (1.4)).
  for every tracker of both layer **do**
    if $\mathcal{A}_i > \mathcal{A}_{up}$ **then**
      Upgrade tracker to visible layer.
    else if $\mathcal{A}_i < \mathcal{A}_{down}$ **then**
      Delete tracker i
    else
      Downgrade tracker to hidden layer, and keep the same parameters and position.
    end if
  end for
  for every pair of trackers of both layers **do**
    Update tracker position using repulsion equation (1.13)
  end for
  for every tracker of hidden layer **do**
    Update tracker position using attraction equation (1.14)
  end for
end for

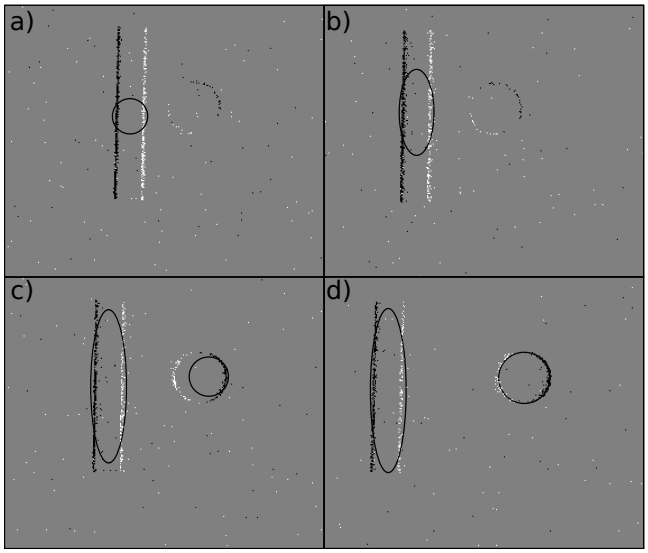


Figure 1.5: Events cloud accumulated over 15ms (for each subfigure), ON and OFF events are represented by white and black points respectively. Gaussian trackers are attracted and deformed in response to the distribution of events clouds corresponding to different objects. (a) The activity of a hidden tracker crosses the $\mathcal{A}_{up}$ threshold, it then becomes visible. (b) The tracker rapidly adapts to the events cloud's distribution. (c) While the first tracker matches the rectangle object, another object starts to move, causing a second tracker to update (d).

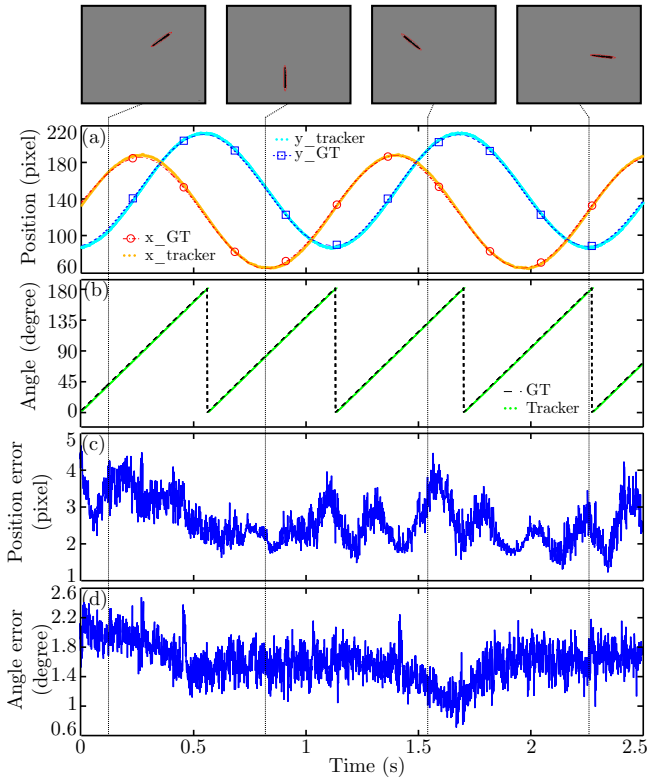


Figure 1.6: Gaussian tracker accuracy in position and angle. (a) Object position in x (dashed line with circle markers) and y (dashed line with square markers). Underlying dots represent the tracker's positions. They are hardly distinguishable from the GT measure. (b) Object's angle (dashed line) and tracker's angle (underlying dots). (c) and (d) Tracking error respectively in position and angle computed every millisecond. On top are shown snapshots (accumulation of events during 10ms) of the segment at different timestamps during motion and the ellipse representing the active Gaussian tracker's position, size and orientation.

linux. We experimentally set the parameters related to the activity of trackers to $A_{up} = 0.15$ and $A_{down} = 0.1$. Parameters related to the extraction of orientations from the covariance matrix have been set to $K = 1$ [see (1.7),(1.8)] and $\tau = 20$ ms (1.6). The repulsion and attraction rates have been set to $d_{max} = 40$ (Eq (1.14)) and $d_{rep} = 20$ (Eq (1.13)). These have been set according to the mean size of observed objects. The repulsion and attraction rates have been set to $\alpha_{att} = 0.01$ (Eq (1.14)) and $\alpha_{rep} = 0.1$ (1.13). Parameters $\tau = 5ms$ (1.10) and $\tau_{inhib} = 10\mu s$ (1.11) have low values so that trackers adapt to fast motions.
As in every low level image processing technique, these values can unfortunately only be experimentally adjusted from the statistics of observed scenes. We found these values to be optimal for all performed experiments. They appear to comply with a large variety of conventional indoor environments for fast and slow stimuli.

1.4.1 Gaussian blob trackers

Fig. 1.5 shows the Gaussian trackers growth and adaptation to events clouds for two independently moving objects (a rectangle and a disc, note that the rectangle only produces two lines of events because it moves in a parallel direction to its two missing edges). The settings of parameters were adjusted experimentally to: $\alpha_1 = 0.2$ (Eq (1.3)), $\alpha_2 = 0.01$ (Eq (1.4)). The adjustment factors have been set smaller for the covariance matrix so that the trackers' size does not collapse in case few events are generated by the target. In practice, the value of the position of the tracker has to move fast to keep up with the motion, while we assume that the shape of the object will change at a slower rate.

We evaluated the Gaussian tracker accuracy, for position, angle and size. In a first experiment, the object tracked is a segment rotating along a circle at constant speed. During ten rotations, only one Gaussian tracker activated and followed the segment. As shown in Fig. 1.6, ground truth (GT) position in x and y (respectively lines with circle and square markers) are barely distinguishable from the tracker's position which is output every millisecond for display purpose but internally updated asynchronously every time an event occurred. The position tracking error is computed as the euclidian distance between the segment's ground truth position and the tracker's position every millisecond: (2.61 ± 0.62 *pixels*, $mean \pm STD$). The segment angle (dashed line Fig. 1.6 (b)) is also tracked (underlying points) with a very high precision (1.59 ± 0.26 degree). Due to symmetries, angles are shown from 0 to 180 degree. Ground truth's position, angle and scale were measured by hand-clicking the segment's extremities periodically on reconstructed frames.

To measure the tracking error using the Gaussian tracker, we moved the same line segment closer and further in front of the retina so that its length changes as we moved closer and farther away. Fig. 1.7 shows the ground truth evolution of the length of the line segment (black dashed line) and the corresponding Gaussian tracker's major axis (small dots). The corresponding tracking error (bottom curve) is 1.9 ± 1.6 pixel.

Finally we performed an experiment where a pen is thrown in front of the camera. The result of its real-time tracking is shown in Fig. 1.8. The pen is successfully tracked in real-time. Some trackers are activated by the person's motion, such as his head and hand that are then also tracked. The mean error of tracking is 1.2 pixel, it corresponds to the distance between the gravity center of the pen and that of the tracker. The ground truth has been performed by manually labeling the gravity center of the pen on reconstructed frames.

1.4.2 Gabor kernels

All Gabor trackers size have been initially set to $\sigma = 3$ (Eq (1.12)) with $\gamma = \sigma/15$ (Eq (1.12)) and $\lambda = 4\sigma$ (Eq (1.12)). The kernel for each orientation has a size of 17×12 pixel and is constant through the experiment (so the size and orientation of each filter will not change during tracking). Four Gabor oriented kernels (see Fig. 1.9) are used to track the thrown pen. As shown by Fig. 1.9, the oriented trackers successfully track the rotating pen. From a handmade ground truth we found that the mean distance between the position of tracked oriented edges and the center of trackers has a mean value of 1.4 pixel. The oriented filters show a 100% of success in detecting orientation.

1.4.3 Gabor combinations and general kernels

In order to evaluate the spatio-temporal precision of the algorithm, we tested it with more complex kernels. We used combinations of Gabor functions (crosses, Fig. 1.3 (e) and (f)) and more general

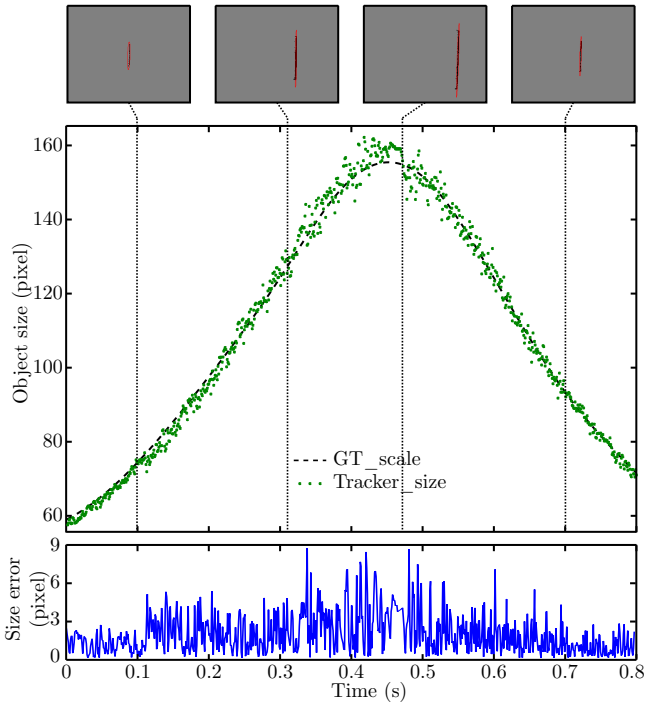


Figure 1.7: Gaussian tracker accuracy in size. (a) Object's ground truth size (black dashed line), and Gaussian tracker's major axis (underlying dots). (b) Size's error computed every millisecond. On top are shown snapshots (accumulation of events during 10ms) of the segment at different timestamps during motion. The Gaussian tracker is shown as en ellipse.

handmade kernels (triangle and square Fig. 1.3 (i) and (j)) were used with a 6×3 initialization grid of hidden trackers for each type of kernel. As in previous experiments, active trackers are represented in Fig. 1.10 by corresponding superimposed shapes. The stimuli were printed on a sheet of paper that was held by hand and moved in front of the sensor. During the experiment, one unique tracker per feature was active, it successfully followed its corresponding shape.

Fig. 1.11 shows the spatiotemporal precision of the algorithm, showing the ground truth (line with circle markers) and the corresponding active tracker's positions over time (small dots). The ground truth is computed every 10*ms* (line with circle markers) accumulating OFF events between two measures and hand-clicking the feature center. We used a single polarity to provide a precise measurement of the accuracy of the trackers when following a single contour. In order not to overload Fig. 1.11, the tracker's position is showed every 1ms. The activities and positions are updated asynchronously every time an event occurs.

The tracking error is computed as the euclidian distance between the tracker's position and the ground truth. It is shown in Fig. 1.12 for all used kernels. The two crosses kernels show a low error of 0.81 ± 0.42 pixel and 0.70 ± 0.38 pixel (mean $\pm$ STD). The error of the square kernel is a bit higher (1.12 ± 0.66 pixel) but remains lower than one pixel. However, the last kernel (triangle) has a significantly higher tracking error. This is again due to the event-based acquisition mechanism. When the relative motion is parallel to a segment of the shape, no event is acquired from this segment. As shown in Fig. 1.13, during vertical motion, the vertical segments of the

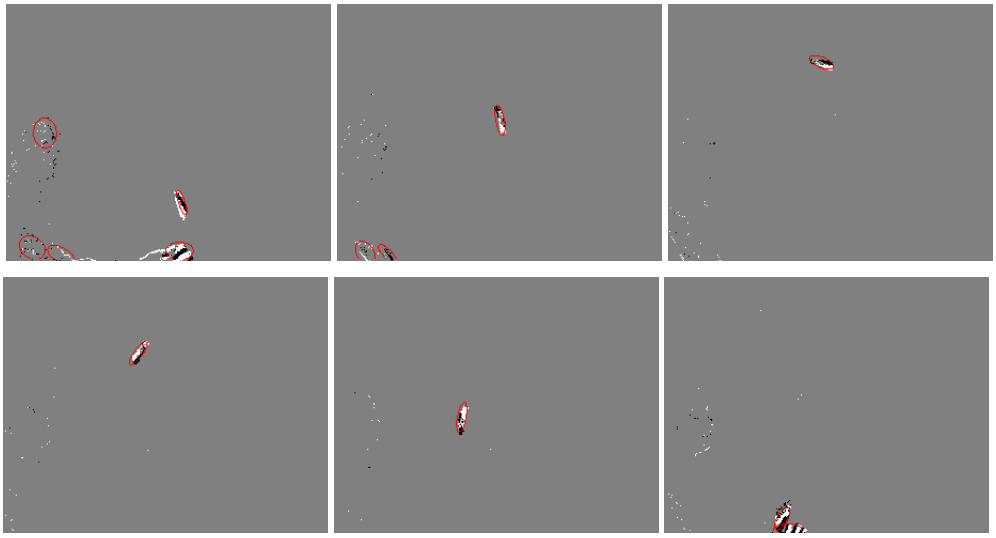


Figure 1.8: From top to down and left to right: sequence of snapshots where the Gaussian blob trackers follow a pen thrown in the air. The trackers stick to the pen's position. Blobs are also activated when the person on the left of the scene is moving. They are successfully tracking the person's features and eventually disappear in absence of motion.

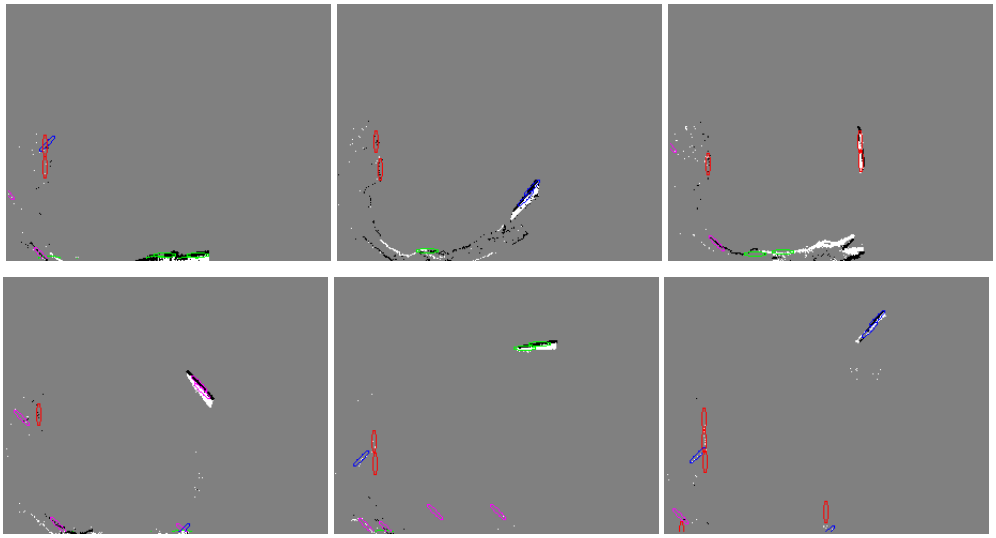


Figure 1.9: From top to down and left to right: sequence of snapshots where oriented Gabor trackers follow the rotation of a pen thrown in the air. The orientation of the pen is successfully tracked. This experiment uses four pools each containing one particular Gabor tracker with a given size and orientation (though tracking a total of four different orientations independently). Oriented trackers are activated sequentially while the pen performs a full 360° rotation. The orientation filters are also activated by the person's movements when they happen. Note how the kernels track the pen but never overlap. This is due to the repulsion mechanism.

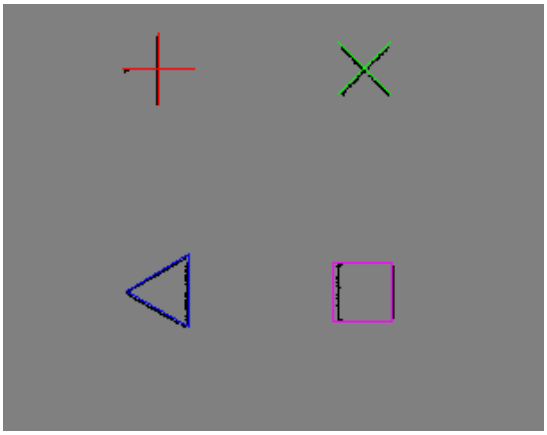


Figure 1.10: Tracking results for specific features using combinations of Gabors and general kernels. Each active kernel is represented by the corresponding shape. The snapshot has been created accumulating events (black dots) during 10*ms*.

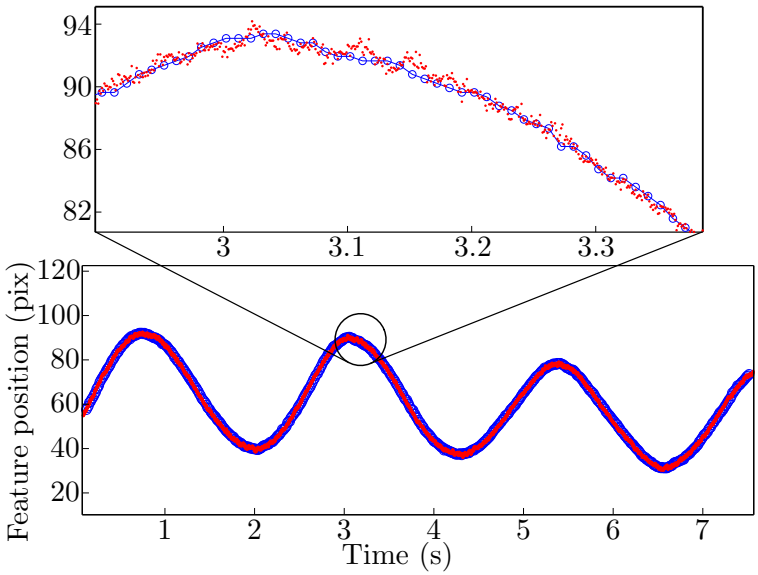


Figure 1.11: Horizontal position of the first active tracker (straight cross in top left corner of Fig. 1.10) over time (small dots), and the ground truth position of the tracked shape (line with circle markers).

triangle, rectangle and straight cross disappear.

Fig. 1.13 shows the motion of the object (dashed circle), and the different positions (segments along the circle). These positions are also reported on Fig. 1.12 and correspond to the configurations that lead to an increase of the triangle tracking error (bottom plot).

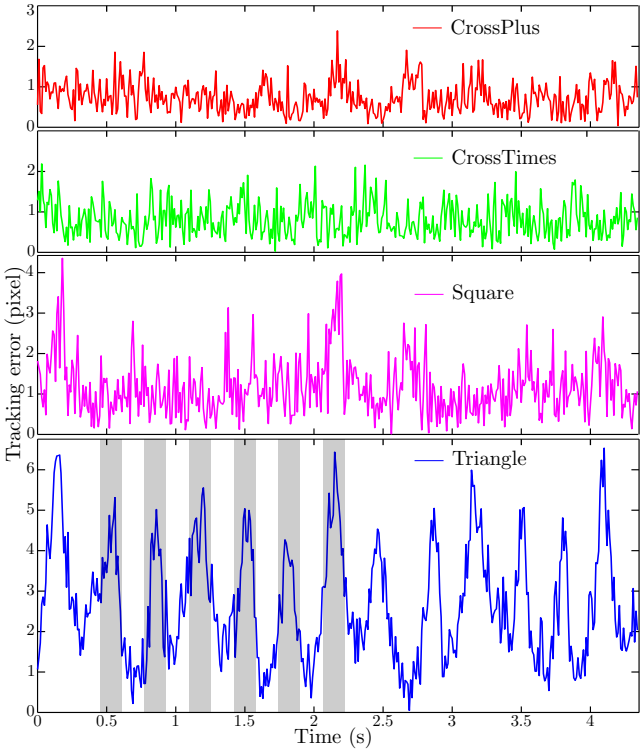


Figure 1.12: Tracking errors for all used features. The error is computed as the euclidian distance between the tracker's position and the ground truth position of the corresponding shape. Gray areas show the periods of time of a single rotation. When the motion direction is parallel to one of the triangle's segments, this leads to an increase of the tracking error. This is an expected result.

1.4.4 Computational cost

We estimated the average time per event spent to update each tracker by measuring the total computation time required by the algorithm in experiments showing a uniform event rate when using a single computation thread. On the computer used in the experiments (Intel Xeon E5520 at 2.27 GHz), this time is approximately 25*ns* per event and per tracker for a kernel size of 70x70 if the kernel's matrix has been pre-computed and processing regulation steps (trackers' repulsion/attraction every millisecond) when running on a single core. Consequently, in a typical natural scene (that statistically generates 200000 events per second), we estimate that it is possible to compute in real-time around 200 of these kernels with one single core of our cpu. The total performance of a multi-threaded implementation using all of the cpu's cores has not been verified experimentally.

The Gaussian trackers are obviously the most computationally expensive ones due to the need to provide a value of the exponential. For the other filters, the computation costs are similar. We precomputed the kernel matrices changes for every position of the incoming event in the kernel. This turns out to be computationally inexpensive as the cost becomes that of accessing a lookup table in memory.

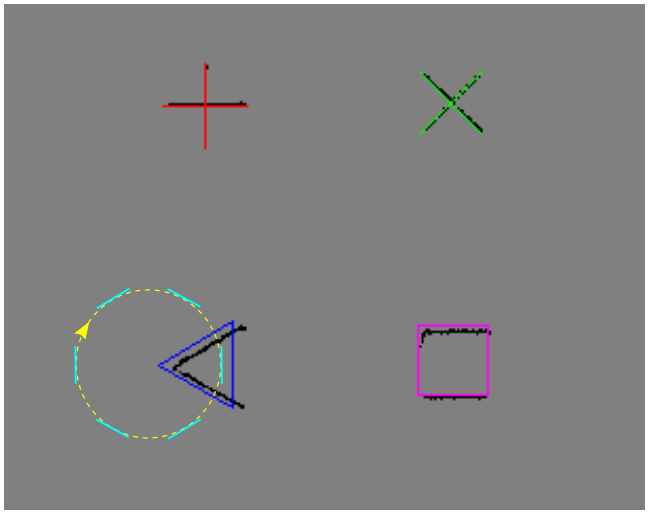


Figure 1.13: Active trackers' positions (straight shapes) and the corresponding ground truth (black dots). Dashed circle indicates the approximate triangle's motion. Segments on this circle represent the positions where motion is parallel to one side of the triangle, leading to an increase of tracking error.

1.4.5 Retrieving feature scale and orientation

Feature trackers use kernels with fixed size and orientation but it is also possible to track a feature even if its size and orientation evolve during time by computing in parallel multiple layers of trackers. Each layer behaves as described in previous sections. This allows to track features' size and orientation in a discretized space. In Fig. 1.14 we show the continuous tracking of a cross that rotates around its center at a constant speed. Due to the cross radial symmetry, we discretized the orientation space into 10 orientations from 0 to 180 degree and also added the ground truth.

The orientation of active tracker regularly alternates to follows the feature's orientation. We can in principle merge the 10 oriented trackers used here into a single rotation-invariant tracker for this cross by computing a global activity as the maximum of each oriented tracker activity. Similarly, we can create scale-invariant feature trackers by combining multiple trackers whose kernels represent a unique feature at different scales. Fig. 1.15 shows the continuous tracking of a square that was moved back and forth in front of the camera. Six scaled square trackers were used corresponding to sizes of 16, 32, 48, 64, 80 and 96 pixels.

1.5 Conclusion and Discussion

This chapter presents an event-based methodology to track shapes. It allows the following of specific shapes at a very low computational cost thus matching the high temporal resolution of event-based vision sensors. It can adapt to orientation and change of scale. Experiments have shown the stability and repeatability of the algorithm. Because of its low computational cost, the method can track multiple targets in parallel. The same technique is also used to make the algorithm scale and orientation invariant. The method is particularly adapted to applications such as robotics vision based navigation, Simultaneous Localization And Mapping (SLAM) or object

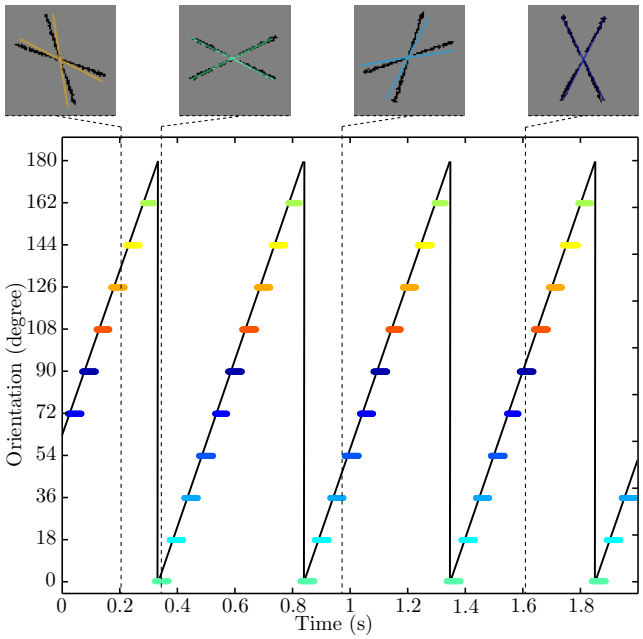


Figure 1.14: Continuous tracking of a rotating cross. Each point on the lower plot represents the activation of an oriented tracker. Its orientation is represented by its level. The orientation's ground truth (black line) was measured every 100ms and interpolated considering constant rotational speed. Upper images represent accumulations of OFF events during 10ms at different timestamps and corresponding active trackers. Only OFF events are used here to ensure a precise measurement of trackers to a single contour.

recognition. These usually impose tight constraints on computation times and energy consumption.

The method relies on several parameters that need to be tuned. This is generally the case for almost every image processing method. Free-parameters techniques in computer vision is still an open problem. In this work we provided a set of parameters that we have shown to be efficient for a wide variety of applications including highly dynamic scenes.
This work is currently being extended to link trackers together in a coherent spatial arrangement to match complex shapes such as faces and human body. These techniques are known as part-based approaches [47], they are currently too computationally greedy in the conventional frame-based approach to be implemented at several kilohertz. Finally, it is important to emphasize the particular innovation of the method. The architecture avoids the N^2 operations per event associated with conventional kernel-based convolutional operations with $N \times N$ kernels. Our transfer function based approach enables us to compute (or retrieve from a look-up table) only one value per tracker for each incoming event which is dependant on the position of the event with respect to the center of the trackers. This is a major feature as, due to the number of incoming events, a full convolution would prevent from real-time implementation on current off-the-shelf computers.

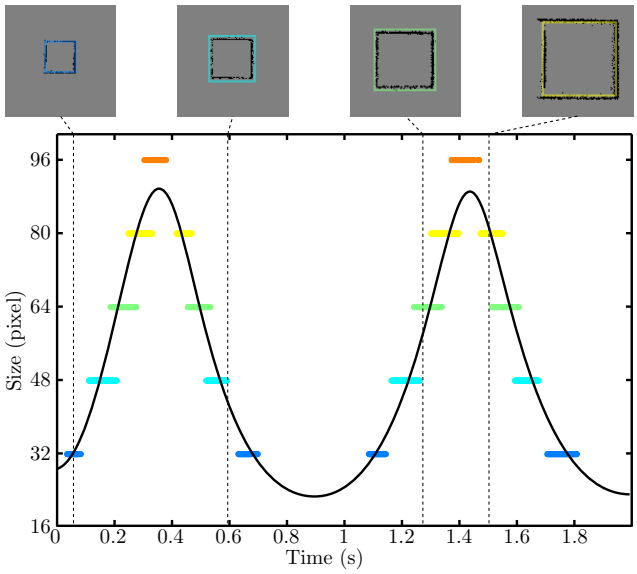


Figure 1.15: Continuous tracking of a moving square. Each point on the lower plot represents the activation of a specific scale tracker. Its size is represented by its level. The Ground truth (black line) was measured every 10*ms*. Upper images represent accumulations of events during 10ms.

Chapter 2

Spatiotemporal Features for Asynchronous Event-based Data

Approaches for higher-level computer vision often rely on the realiable detection of features in visual frames, but similar definitions of features for the novel dynamic and event-based visual input representation of silicon retinas have so far been lacking. This chapter addresses the problem of learning and recognizing features for event-based vision sensors, which capture properties of truly spatiotemporal volumes of sparse visual event information. A novel computational architecture for learning and encoding spatiotemporal features is introduced based on a set of predictive recurrent reservoir networks, competing via winner-take-all selection. Features are learned in an unsupervised manner from real-world input recorded with event-based vision sensors.

Contents

2.1 Introduction

Humans learn efficient strategies for visual perception tasks by adapting to their environment through interaction, and recognizing salient features. In contrast, most current computer vision systems have no such learning capabilities. Despite the accumulated evidence of visual feature learning in humans, little is known about the mechanisms of visual learning [48]. A fundamental question in the study of visual processing is the problem of feature selection : which features of a scene are extracted and represented by the visual cortex? Classical studies of feature selectivity of cortical neurons have linked neural responses to properties of local patches within still images [49, 46]. Conventional artificial vision systems rely on sampled acquisition that acquires static snapshots of the scene at fixed time intervals. This regular sampling of visual information imposes an artificial timing for events detected in a natural scene. One of the main drawbacks of representing a natural visual scene through a collection of snapshot images is the complete lack of dynamics and the high amount of redundancy in the acquired data. Every pixel is sampled continuously, even if its output value remains unchanged. The output of a pixel is then unnecessarily digitized, transmitted, stored, and processed, even if it does not provide any new information that was not available in preceding frames. This highly inefficient use of resources introduces severe limitations in computer vision applications, since the largely redundant acquired information lead to a waste of energy for acquisition, compression, decompression and processing [11].

Biological observations confirm that still images are largely unknown to the visual system. Instead, biological sensory systems are massively parallel and data-driven [50]. Biological retinas encode visual data asynchronously through sparse firing spike trains, rather than as frames of pixel values [51]. Current studies show that the visual system effortlessly combines the various features of visual stimuli to form coherent perceptual categories relying on a surprisingly high temporal resolution: the temporal offsets of on-bistratified retina cells responses show an average standard deviation of 3.5ms [52, 53]. Neurons in the visual cortex also precisely follow the temporal dynamics of the stimuli up to a precision of 10ms.

In order to bridge the gap between artificial machine vision and biological visual perception, computational vision has taken inspiration from fundamental studies of visual mechanisms in animals [46, 54]. One main focus of these approaches have been various computational models of simple and complex cells in the primary visual cortex (V1) [46, 55, 56], which are characterized by their preferred response to localized oriented bars. Typically, this orientation-tuned response of V1 cells has been modeled with Gabor Filters [57], which have been used as the first layer of feature extraction for visual recognition tasks [58, 59]. The most well-known example of biologically inspired, although still frame-based model of object recognition is the HMAX model [56, 60, 61]. It implements a feedforward neural network based on a first layer of Gabor filters followed by different layers realizing linear and nonlinear operations modeled on primate cortex cells. However, HMAX like other approaches implementing neural networks to perform visual tasks [62] are still based on processing still images and therefore cannot capture key visual information mediated by time.

This chapter introduces an unsupervised system that allows to extract visual spatiotemporal features from natural scenes. It does not rely on still images, but on the precise timing of spikes acquired by an asynchronous spike-based silicon retina [11]. The development of asynchronous event-based retinas has been initiated by the work of Mahowald and Mead [1]. Neuromorphic asynchronous event-based retinas allow new insights into the capabilities of perceptual models to use time as a source of information. Currently available event-based vision sensors [10, 13]

produce compressed digital data in the form of time-stamped, localized events, thereby reducing latency and increasing temporal dynamic range compared to conventional imagers. Because pixel operation is now asynchronous and pixel circuits can be designed to have extremely high temporal resolution, silicon retinas accomplish both the reduction of over-sampling of highly redundant static information, as well as eliminating under-sampling of very fast scene dynamics, which in conventional cameras is caused by a fixed frame rate. Pixel acquisition and readout times of milliseconds to microseconds are achieved, resulting in temporal resolutions equivalent to conventional sensors running at tens to hundreds of thousands of frames per second, without the data overhead of conventional high-speed imaging. The implications of this approach for machine vision can hardly be overstated. Now, for the first time, the strict temporal resolution vs. data rate tradeoff that limits all frame-based vision acquisition can be overcome. Visual data acquisition simultaneously becomes fast and efficient. A recent review of these sensors can be found in [10, 63].

Despite the efficiency of the sensor representation, it is far from straightforward to port methods that have proven successful in computer vision to the event-based vision domain. Much of the recent success of computer vision comes from the definition of robust and invariant feature or interest point extractors and descriptors [64, 65, 66]. Although such methods have proven to be very useful for static image classification, they require processing of the whole image, and do not take temporal information into account. Dynamical features for event data should instead recognize features only from novel visual input, and recognize them as they appear in the sparse input stream. This requires a model that can continuously process spiking inputs, and maintain a representation of the feature dynamics over time, even in the absence of input. Here we present an architecture for feature learning and extraction based on reservoir computing with recurrent neural networks [67], which integrate event input from neuromorphic sensors, and compete via a Winner-Take-All (WTA) technique to specialize on distinct features by predicting their temporal evolution.

A proof of concept for the performance of the architecture is demonstrated in three experiments using natural recordings with event-based vision sensors. In the first experiment, we present a set of oriented bars to the camera in order to the show the capacity of the model to extract simple features in an unsupervised manner, using a big spatial receptive field to emphasize the graphical visualization of the learnt features. In the second experiment, the full capacity of the method is demonstrated by mapping the field of view to several small receptive fields, and showing that the model is still capable of reliably extracting features from the scene. The last experiment applies the architecture to complex object features. All experiments were conducted with real-world recordings from DVS cameras [11], and thus are subject to the standard noise distribution of such sensors.

2.2 Material and Methods

2.2.1 Event-based asynchronous sensors

In our experiments we used asynchronous event-based input signals from a Dynamic Vision Sensor (DVS) [11], whose pixels are based on the same principle as the change detectors of the ATIS camera [13][12] described Section 1.2. It encodes visual information using the Address-Event Representation (AER), and has a spatial resolution of 128×128 pixels. We define an event occur-

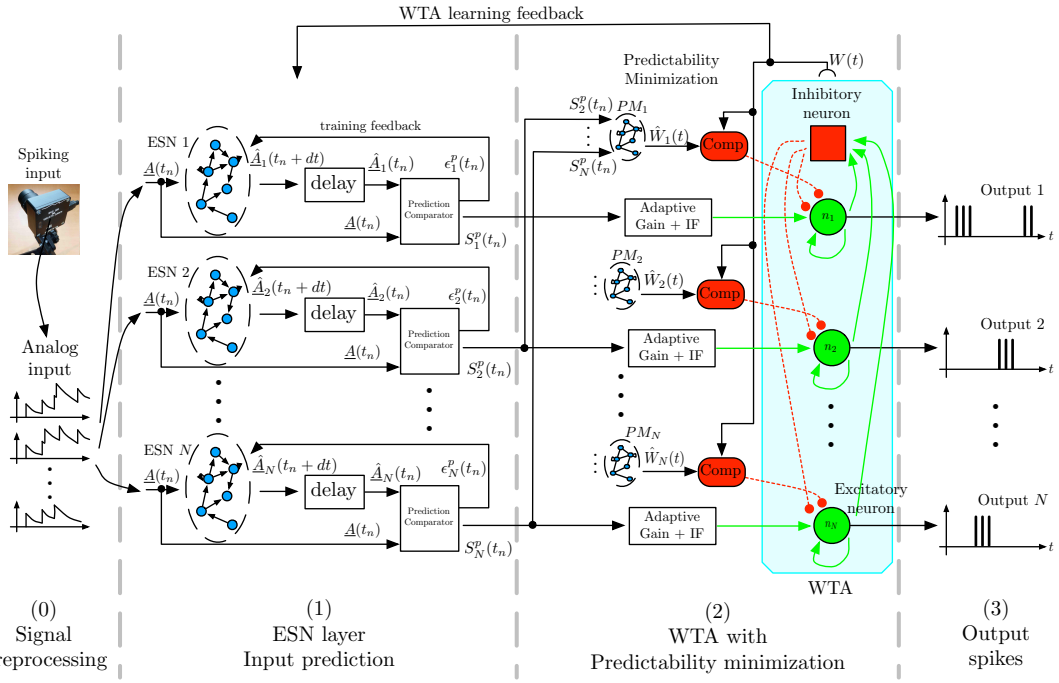


Figure 2.1: Architecture for unsupervised spatiotemporal feature extraction: spikes from the DVS are transformed by filtering into analog input signals that are sent to a set of ESN networks. Each ESN is trained to predict future input activations based on current and past activities. The prediction is compared to the actual inputs, and the output signal S_k^p, which is a representation of the ESN's prediction performance is fed into a Winner-Take-All (WTA) network. This WTA selects the best predicting ESN and enables it to train on the present input sequence. A predictability minimization process promotes orthogonality of predictions between the different ESNs during the WTA selection. The combination of temporal prediction and competition through the WTA allows each ESN to specialize on the prediction of a distinct dynamical feature, which thus leads to learning of a set of different feature detectors.

ring at time t at the pixel $(x,y)^T$ as:

$$e(x,y,t) = |p| = 1 \quad , \tag{2.1}$$

where p is the polarity of the event. p equals 1 ("ON") whenever the event signals an intensity increase, or -1 ("OFF") for a decrease, but for the purposes of this work the polarity is not used.

2.2.2 General architecture

Fig. 2.1 shows the general architecture of the feature selection process. In the following we briefly describe the overall architecture, with more detailed descriptions of the individual components below. To capture the temporal dynamics of spatiotemporal features, we use Echo-State Networks (ESN) [68] that act as predictors of future outputs. To achieve unsupervised learning of distinct features we use multiple ESNs that compete for learning and detection via a WTA network. As

the first stage, the signal coming from the DVS retina is preprocessed, by converting the DVS output into analog signals as required by the ESNs' structure. In the second stage, labeled *ESN layer* in Fig. 2.1, each ESN receives the converted output of the DVS to predict its evolution one timesteamp in the future. The readout of each ESN is trained for this task, and each network should learn to predict different temporal dynamics. To achieve this, the next layer of the architecture, labeled *WTA with Predictability minimization* in Fig. 2.1, implements a Winner-Take-All (WTA) neural network, which selects the best predictor from the available set of predicting ESNs. Through competition, the WTA inhibits poorly predicting ESNs to ensure that the best predictor has sufficient time to learn a particular spatiotemporal sequence. This layer also contains a predictability minimization process to promote orthogonality of predictions between the different ESNs. The selected ESN is then trained to recognize the spatiotemporal pattern, and learns to predict its temporal evolution. The WTA competition ensures that each ESN specializes on an independent feature, thus preventing two ESNs from predicting the same pattern. Consequently, at any given time, the winning network in the WTA layer indicates the detected feature. Through random initialization of ESNs and WTA competition, the architecture extracts distinct spatiotemporal features from event-based input signals in a completely unsupervised manner.

For the experiments described in this article, the architecture has been fully implemented in software, using DVS recordings of real-world stimuli as inputs. In particular, the visual inputs for all experiments contain the typical noise for this kind of sensor, and do not use idealized simulated data.

2.2.3 Signal pre-processing

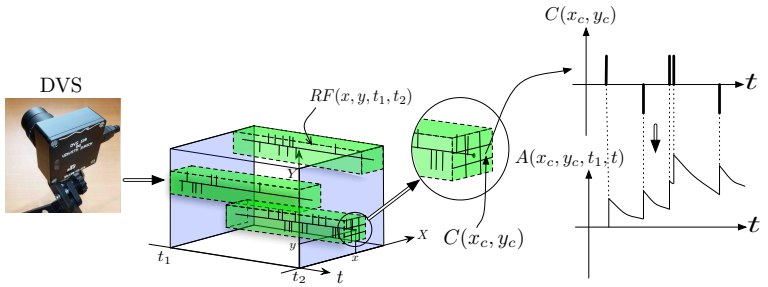


Figure 2.2: Illustration of signal pre-processing to convert DVS events into equivalent analog input for ESNs. In order to reduce the number of input channels to the system and reduce computational load, the input from retina pixels is first spatially subsampled into cells $C(x_c,y_c)$. Each set of ESNs then receives input from a particular receptive field $RF(x,y,t_1,t_2)$. To compute the equivalent analog input, an exponential kernel finally is applied to each event contained in the receptive field.

The DVS retina has approximately $16K$ pixels in total. Directly using each pixel as an input to the ESN reservoir would require a network with $16K$ input neurons, and, in typical reservoir computing setups, 10 to 100 times more hidden neurons. Since this is a prohibitevily large size for real-time simulation of neural networks on conventional current computers, a pre-processing stage is introduced to downsample the dimensionality of the input. Please note that this is not a fundamental requirement, since especially future large-scale neuromorphic processors and other

dedicated hardware platform could potentially handle real-time execution of such large networks (see Discussion), but this is beyond the scope of our proof-of-principle study.

Fig. 2.2 provides a more detailed view of the first layer of the architecture, named layer (0) in Fig. 2.1. To reduce the input dimensionality of the DVS signal, the retina pixels are first spatially resampled into cells $C(x_c, y_c)$ of $\delta_x \times \delta_y$ pixels, each integrating pixels around the center (x_c, y_c) according to:

$$C(x_c, y_c) = \left\{ (x,y) \quad | \quad \begin{array}{l} x \in [x_c - \delta_x, x_c + \delta_x] \\ y \in [y_c - \delta_y, y_c + \delta_y] \end{array} \right\}. \tag{2.2}$$

Next, the signals are quantized by introducing spatiotemporal receptive fields $RF(x_0, y_0, t_1, t_2)$, covering $\Delta_x \times \Delta_y$ subsampling cells, which collect all events in a spatiotemporal volume in the time interval $[t_1, t_2]$ according to:

$$RF(x_0, y_0, t_1, t_2) = \left\{ e(x,y,t) \quad | \quad t \in [t_1, t_2], \quad (x,y) \in C(x_c, y_c), \quad \begin{array}{l} |x_c - x_0| \leq \Delta_x \\ |y_c - y_0| \leq \Delta_y \end{array} \right\}. \tag{2.3}$$

Conversion of events into analog signals is achieved by filtering with a causal exponential filter with time constant τ, defined as $G(t, t_i) = e^{-(t-t_i)/\tau} \cdot H(t - t_i)$, where $H(t)$ is the Heaviside function, which is 1 for $t \geq 0$ and zero otherwise. This filter is applied to all spikes coming from pixels (x, y) contained in a receptive field $RF(x_0, y_0, t_0, t)$, yielding the analog output signal A, which is fed into the ESNs:

$$A(x, y, t_0, t) = \sum_{\substack{e(x_i, y_i, t_i) \in RF(x_0, y_0, t_0, t) \\ x_i = x, \; y_i = y}} G(t, t_i) \quad , \tag{2.4}$$

where t_0 is a chosen time origin.

The complete preprocessed input at time t fed into the ESN layer is the vector formed by all outputs $A(x, y, t_0, t)$ of pixels contained in $RF(x_0, y_0, t_0, t)$. For clarity, we will in the following only consider a single receptive field denoted as $\underline{A}(t)$:

$$\underline{A}(t) = \begin{pmatrix} A(x_1, y_1, t_0, t) \\ \vdots \\ A(x_M, y_M, t_0, t) \end{pmatrix} \tag{2.5}$$

2.2.4 ESN layer – Input prediction

This layer (Fig. 2.1-(1)) computes the prediction of input signals for N different ESNs [68]. The k^{th} ESN is defined by its internal state s^k, and the three weight matrices W^k_{out} (for output or *read-out* weights), W^k_{in} (for input weights), W^k_{back} (for feedback weights), and the recurrent weights W^k_{r}. These weight matrices are initialized randomly for each ESN and encoded as 64 bit floating-point numbers. The internal state s^k of the ESN and its output (out^k) are iteratively updated, and evolve according to :

$$s^k(t_n) = f\left(W^k_{\text{r}} \cdot s^k(t_{n-1}) + W^k_{\text{in}} \cdot \underline{A}(t_n) + W^k_{\text{back}} \cdot out^k(t_{n-1}) \right) \quad , \tag{2.6}$$

$$out^k(t_n) = f^{\text{out}}\left(W^k_{\text{out}} \cdot s^k(t_n) \right) \quad . \tag{2.7}$$

In our experiments, the logistic function is used as the nonlinearity f for the internal state evolution, and a linear readout is used as f^{out}. Every ESN is trained to predict its future input at one timestep ahead (i.e. at $t_n + dt$), thus the output of the ESN according to Eq. 2.7 creates a prediction $\hat{\underline{A}}_k(t_n + dt) = out^k(t_n)$, which should match $\underline{A}(t_n + dt)$. As is usual for ESNs, only the readout weights W^k_{out} are adapted, the recurrent and other weights are kept at their random initial values which are drawn from uniform distributions.

As suggested in [69], training of the readout weights W^k_{out} can be achieved with a standard recursive least squares algorithm (here a version described in [70] was used). This algorithm recursively adapts W^k_{out} so as to minimize a weighted linear least squares cost function, computed from the prediction error:

$$\varepsilon^p_k(t_n) = |\hat{\underline{A}}_k(t_n) - \underline{A}(t_n)|. \tag{2.8}$$

This method is well suited for online learning, since the coefficients of W^k_{out} can be updated as soon as new data arrives.

The output of the ESN layer into the subsequent WTA layer is a similarity measure $S^p_k(t_n)$ for each ESN, which indicates the quality of each prediction for the currently observed input:

$$S^p_k(t_n) = \frac{\sum_i \left|\underline{A}(t_n)_i . \hat{\underline{A}}_k(t_n)_i\right|}{\sum_i |\underline{A}(t_n)_i| \cdot \sum_i \left|\hat{\underline{A}}(t_n)_i\right|}, \tag{2.9}$$

where i is summing over all components of $\underline{A}(t_n)$ and $\hat{\underline{A}}(t_n)$, which have been properly normalized to take on values between 0 and 1.

2.2.5 Winner-Take-All selection

Based on the indicators of prediction quality $S^p_k(t_n)$ computed by the ESN layer, the third layer of the model (Fig. 2.1-(2)) selects the best predictor among the N ESNs through a WTA mechanism. The WTA network consists of a set of N neurons $\{n_1, \ldots, n_N\}$ plus an inhibitory neuron, which is recurrently and bi-directionally connected with the excitatory neurons, as detailed in [71, 72, 73, 74]. The task of the WTA is to select from the pool of ESNs the one whose prediction best matches the actual dynamics of the present input, and which thus has the highest similarity $S^p_k(t_n)$, as computed by layer (1) in Fig. 2.1.

Inputs to the WTA neurons are generated from the S^p_k values using non-leaky Integrate-and-Fire (IF) neurons, which transform the analog values into spike trains. To make the WTA network more robust to the variations in the similarity measure, a sigmoid function is applied to the S^p_k values to compute the input current fed to the IF neurons:

$$g_{\text{IF}}(S^p_k) = G_{\min} + \frac{G_{\max} - G_{\min}}{1 + \exp(-(S^p_k - x_0)/\lambda)}. \tag{2.10}$$

$G_{\min}$ and $G_{\max}$ define the interval in which the output firing rates of the IF neurons are taking values. They are set experimentally to achieve spike rates spanning from 5kHz to 15kHz. λ sets the selectivity of the sigmoid which is an increasing function of λ (λ has been experimentally tuned to $5.0e^{-5}$ in our experiments). The value of the offset x_0, which is subtracted from the S^p_k is managed by a proportional controller. Its input reference is set such that x_0 approaches the value of S^p_k output by the selected best predictor. This ensures that whatever the current state of the system is, the sigmoid g_{IF} is always centered on the current value of interest, giving the best selectivity possible to detect changes in the best predictor. The update period of this controller is set to 0.5

ms. The index of the spiking neuron from the WTA network then corresponds to the best predictor $W(t)$ satisfying :

$$W(t) = \underset{k\in\{1,\ldots,N\}}{\operatorname{argmax}}\, g_{IF}(S_k^p(t)) \quad . \tag{2.11}$$

The obtained index $W(t)$ is used to drive the learning process of the *ESN layer*. Only the ESN selected by the WTA network (ESN with index $W(t)$) is trained on the input signal. This adaptive WTA achieves good performance in the selection of the best predictor even if the similarity measurement has a large variance (this happens for instance if the system is exposed to a set of very different stimuli).

This setup of the WTA architecture always generates outputs, even if no input is present. This potential inefficiency can be avoided by adding another output layer, which computes a gating function that depends on the global input activity. Using this mechanism, output neurons driven by the output of the WTA will only fire if in addition the input activity is bigger than a defined threshold. The threshold can be either defined on the average event rate, or the average value of $\underline{A}(t_n)$.

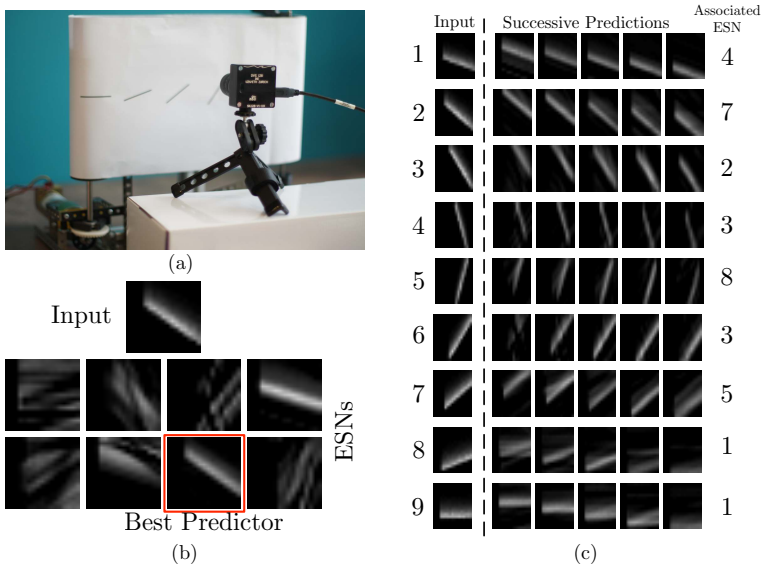


Figure 2.3: Experimental recording setup: (a) A DVS records patterns moving on a treadmill. (b) Current input pattern (top) and predictions of the different ESNs at a given time. The best predictor is highlighted in red. (c) Results of ESN training. The left column shows a snapshot from each of the nine different patterns. The plots to the right show different predictions for different time steps in the future, obtained from the ESN which is specialized in the given pattern. The time difference between the five predicted patterns is 0.01 seconds.

2.2.6 Predictability minimization

The third layer implements, in addition to the WTA selection, a predictability minimization algorithm, which ensures that each ESN specializes in predicting different features in the input. It implements a criterion suggested by [75, 76] to evaluate the relevance of the prediction of each

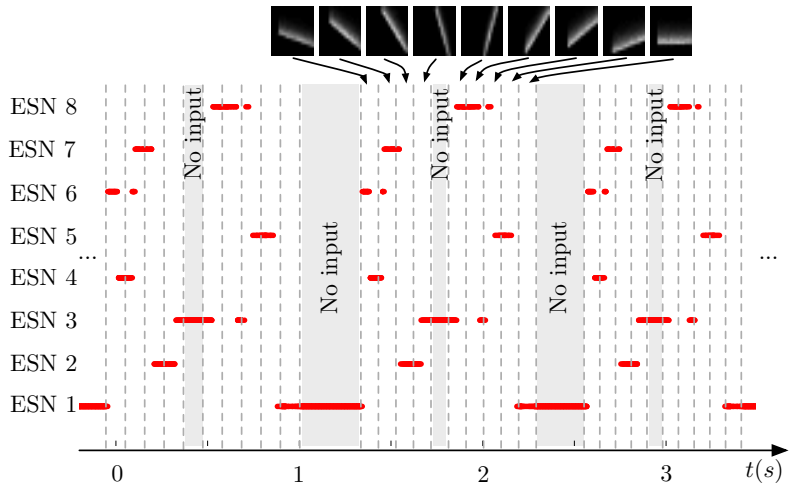


Figure 2.4: Output of the WTA network during repeated presentation of a series of nine input patterns. Red lines indicate when an ESN was selected by the WTA network. Dashed vertical lines mark time points when the input presented to the DVS changed from one pattern to another (the 9 patterns are shown on top of the figure). Shaded areas indicate times when no stimulation was present. Every ESN learns to respond to only a small subset of input patterns (typically exactly one pattern). This response is reproducable over different stimulus presentations.

ESN: an ESN's prediction is considered relevant if it is not redundant given the other ESNs' predictions. This predictability minimization step promotes orthogonality of predictions between the individual ESNs, and encourages a maximally sparse representation of the learned input classes, thereby achieving good coverage of the presented input space. For each ESN k, an estimator $\hat{W}_k$ of the WTA output is used, which receives only the similarity measures $S_{k'}^p$ of the other ESNs as input. For a consistent framework of estimators and predictors, we chose to use ESNs (named $PM_1, \ldots, PM_N$ in Fig. 2.1) to implement the $\hat{W}_k$ estimator. This also allows taking into account the highly dynamic information contained in the input data recorded with the DVS. Training of the ESNs follows the same principles as described in section 2.2.4.

If the estimator $\hat{W}_k$ and the WTA output agree, i.e. $\hat{W}_k(t_n) = W(t_n)$, then this means that the k^{th} ESN is not currently learning a new feature, because the same information can also be deduced from the output of the other ESNs. In this case, the corresponding neuron of the WTA is inhibited to prevent this ESN from learning the currently presented input patter. The inhibition also causes the output of the WTA to stop responding to the input, thus promoting another one.

2.3 Results

2.3.1 Experimental setup

The experiments presented in this article were performed with the setup shown in Fig. 2.3(a). It consists of a DVS retina observing a treadmill, on which moving bars with 9 different orientations move across the field of view of the DVS at constant speed. For the experiments, the recurrent

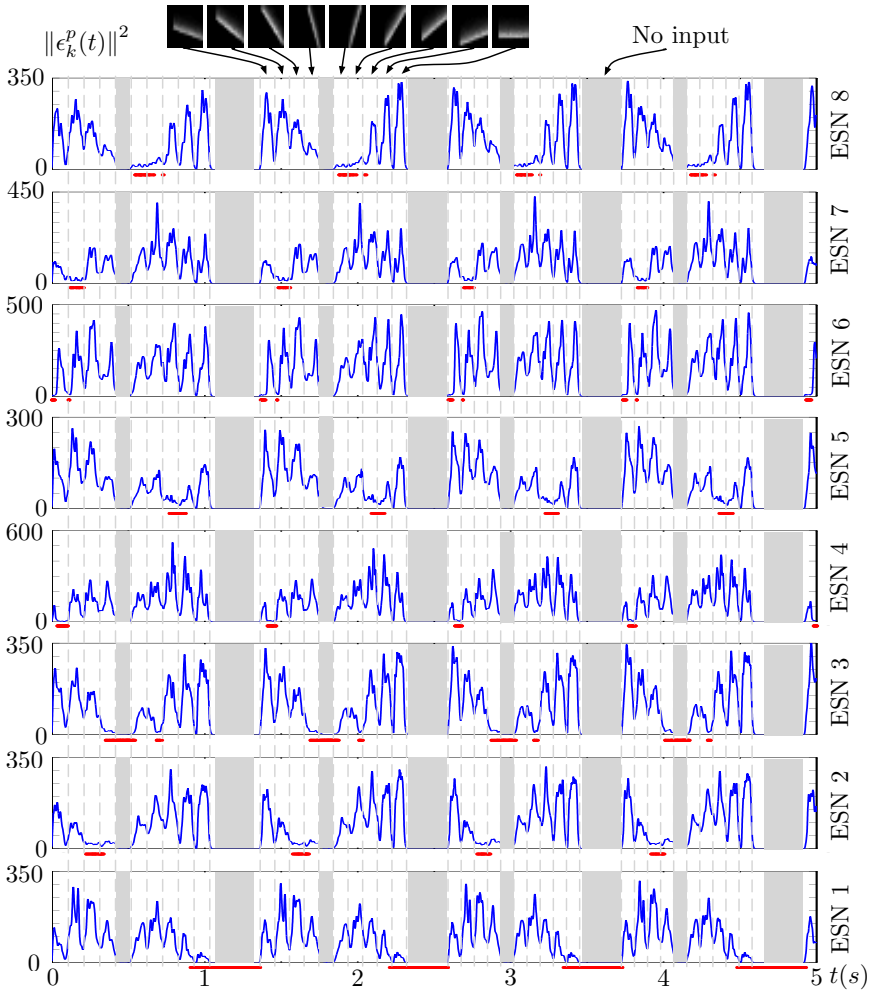


Figure 2.5: Prediction error of the 8 ESNs during several presentations of the input stimulus. Red lines below each plot shows the output of the WTA neuron corresponding to each ESN, thus indicating times when each ESN was selected as the best predictor. We can observe that ESNs are correctly selected when their prediction error is minimal amongst all the networks.

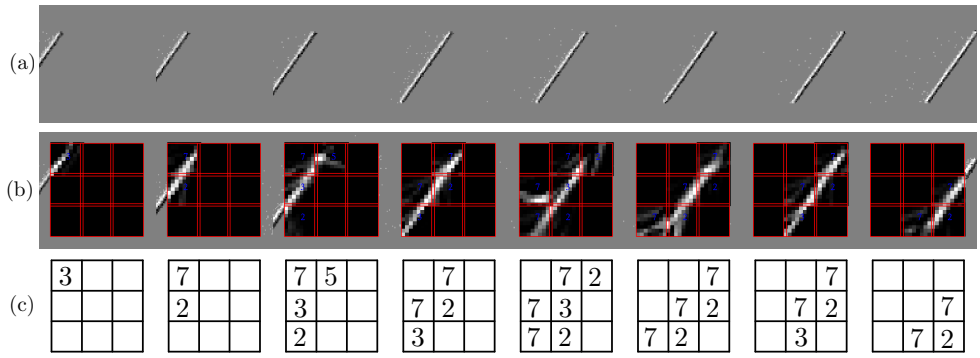


Figure 2.6: Predictions of the ESNs when the camera's field of view is divided into multiple receptive fields (RFs). (a) Output of the DVS for a bar moving across the field of view. (b) Predictions of ESNs processing different RFs (indicated by red boxes). (c) Index of the best predicting ESN in every RF at the timepoints of the snapshots. Indices are only shown if the input activity in the RF exceeds a given threshold.

connectivity matrix W_{r} for each ESN was initialized randomly, and rescaled to have spectral radius 0.7, which fulfills the Echo State Property [68]. The other weight matrices were randomly chosen from a uniform distribution in $[-0.4;0.4]$ for W_{in}, $[-0.02,0.02]$ for W_{back} and $[-0.01,0.01]$ for W_{out}. The pre-processing uses exponential kernels with a time constant of 10 ms.

2.3.2 Single receptive field

The first experiment uses 8 ESNs, each composed of 15 analog neurons, randomly connected in the reservoir. Only one RF, consisting of 17×17 cells $C(x_c, y_c)$ spanning 5×5 pixel is used as input to each ESN. Fig. 2.3(b) shows the different predictions of the ESNs in response to an input signal. The WTA succeeds in selecting the best predicting network for the current input. Fig. 2.3(c) shows for each stimulus the best predictions and the associated ESN. As expected, the results confirm that every network has specialized in the prediction of the temporal evolution of a specific oriented moving pattern. Since natural scenes contain many independent features, which are likely to occur in larger numbers than the number of available ESNs, we tested here the performance of an architecture with only 8 ESNs for 9 different patterns of moving oriented bars. The results indicate that some of the ESNs tend to learn more than one dynamic feature, so that the system can represent all input features as accuractely as possible. In order to select the most appropriate number of predictors, additional control mechanisms could be employed. An example of this is the response of ESN1, which is the best predictor both for pattern 8 and 9 (Fig. 2.3(c)).

Fig. 2.4 shows the output of the same system for three successive testing presentations of the stimulus. We can see that each ESN is responding to a specific orientation of the bars. Moreover, the process is repeatable over the three presentations with a difference in the temporal span of the responses. This is due to the increase of the translation speed of the bars during the recording to show that the networks effectively respond to the bar's orientations independently of their speed.

Fig. 2.5 shows the prediction error of each ESN during several presentations of the stimulus. The output of the WTA network is shown below each curve, indicating when a particular ESN is

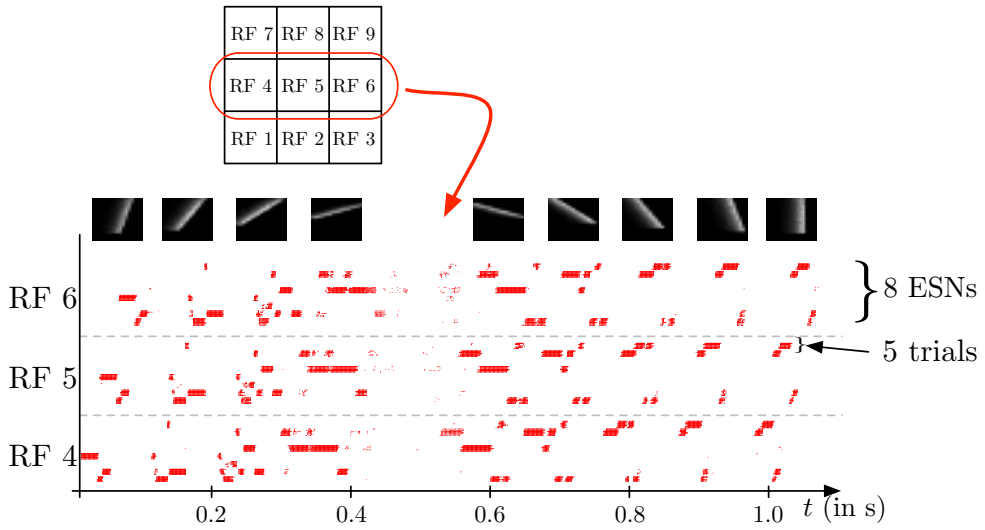


Figure 2.7: Spike output for the WTA neurons corresponding to the eight ESNs for each of the 3 central RFs (see Fig. 2.6) during the presentation of a series of 9 moving bars with different orientations (see snapshots on top). Dots indicate the times of output spikes for 5 repeated stimulus presentations, which are drawn on different coordinates along the y-axis, but grouped by WTA neuron. Each ESN, depending on its index in the RF, is associated with a color used to represent the dots corresponding to its output. The results show a highly reproducable response of the feature detectors for different trials, and also similar time delays for different stimulus presentations. Because the input stimulus moves horizontally, the outputs of the WTA circuits are similar, with a little time delay. Note that only one ESN can be active at any given time in each RF. Apparent simultaneous spikes from multiple WTA neurons are due to the scaling of time in the horizontal axis of the figure.

selected as the best predictor. An ESN is correctly selected whenever its prediction error is the lowest. Periods in which all prediction errors are close to zero correspond to periods without input (shown as gray regions in the figure). This is a result of the approximate linearity of the ESNs and their low spectral radius: when only weak input is fed into the network, the ESNs readout output also approaches zero, which results in a low prediction error for times when no stimulus is presented (the only input to the networks then is background noise from the DVS pixels).

2.3.3 Multiple receptive fields

In the second experiment, the field of view of the DVS is split into 3×3 smaller RFs of identical size (9×9 cells of 3×3 pixels), as shown in Fig. 2.6. This shows the full intended behavior of the system as a local spatiotemporal feature detector, in which different features can be assigned to small receptive fields covering the entire field of view of the sensor (instead of being covered by only one big one RF like in Fig. 2.3). For each RF 8 ESNs are used as feature detectors. In the learning phase, they are trained only with the input to the central RF. Subsequently, their weights are copied and the ESNs are used independently for all 9 RFs. Thus, all RFs have ESNs with identical weights (and so detects the same features), but receive different inputs and therefore evolve

independently. Fig. 2.6(a) shows different snapshots of the DVS recording for an oriented bar moving across the field of view. The output of the predictors for each RF is shown in Fig. 2.6(b), while Fig. 2.6(c) indicates for each RF the index of the ESN selected. The figure also shows that ESN predictors are only selected when there is substantial input activity in the RF. As in the previous experiment, dynamic feature selection is reproducable and exhibits precise timing, as shown in Fig. 2.7. Here, only the 3 RFs on the middle line of the input space are shown. Because the input stimulus moves horizontally, the outputs of the WTA circuits are similar, with a little time delay. Using multiple smaller RFs instead of one is also a potential solution to represent more features with a finite set of ESN. The feature descriptor is then a combination of the outputs of all available ESNs, which need to be processed by another layer. This is however, beyond the scope of the present work.

Choosing the right number of ESNs for the feature detection architecture is not always straightfor-

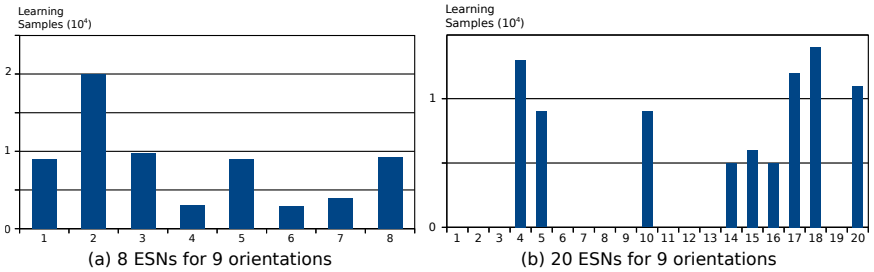


Figure 2.8: Number of learning samples per reservoir for two different architectures applied to the same input. Every step where training of the ESN readout was activated is counted as a learning sample. (a) Learning samples for 8 ESNs trained on 9 different input patterns. (b) Learning samples for 20 ESNs trained on the same 9 input patterns. The results show that when the pool of ESNs is bigger that the number of features present in the input, only a necessary subset of ESNs from the pool is used to learn these features. The remaining ESNs are not trained, and can be used to learn new features from future inputs.

ward, and depends on the number of distinct features present in a scene. In Fig. 2.4 it was shown that when the number of ESNs is smaller than the number of features, an ESN can learn multiple features instead of one. Fig. 2.8(a) shows the number of steps in which each ESN is trained if 8 ESNs are trained on 9 different input patterns. It is shown that all networks are trained for a similar number of epochs. When instead the number of ESNs exceeds the number of features, we find that only the minimum necessary number of predictors is selected, and the remaining ESNs are still available to learn new features, should there be distinct future visual inputs. Fig. 2.8(b) a clear specialization of ESNs, if 20 networks are used to encode the same 9 features that were used in Fig. 2.8(a). Only 9 out the 20 ESNs show increased activation during the stimulation presentations.

2.3.4 Complex input stimulus

In the last experiment, the ability of the architecture to represent more complex features was tested. Instead of using oriented bars, we now present digits (from 1 to 9) to the camera, with a single receptive field covering the whole stimulus. Nine ESNs were used in the system, which matches the number of distinct patterns. To make them visible for DVS recordings, the nine digits were

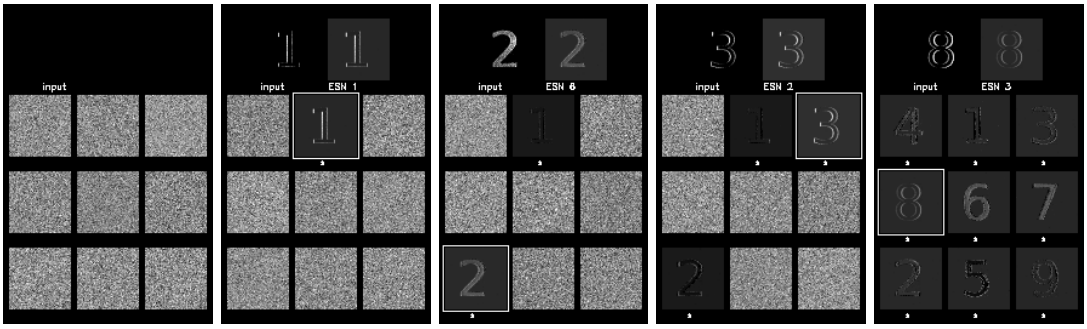


Figure 2.9: Learning of more complex feature detectors, showing the output of the system when presented with stimuli composed of digits from 1 to 9. Each snapshot on the top left shows the analog input to the ESN, obtained by filtering the DVS events. The top right shows the prediction of the best ESN, as selected by the WTA circuit. Below, the current predictions of all nine ESNs used in the experiment are shown. A white square around the prediction indicates the ESN that was selected by the WTA. The snapshots show the progession of learning, starting with an untrained network in (a), which only produces random predictions. (b), (c), (d) show the output of the same networks after the presentation of patterns "1", "2" and "3" (respectively). A white mark underneath an ESN prediction indicates that this ESN has learned a feature. Finally (e) shows its output after the end of the learning process where all networks have learnt an input stimulus.

animated, by hand, with a random jittering movement around a central spatial position. This was intended to simulate what would be seen by the retina when the eye follows microsaccadic movements. Because the jitter is random, the input stimulus mainly contains spatial information. This experiment allows us to test the robustness of the proposed method to several spatiotemporal patterns, including the degenerate case where only one spatial information is relevant for the feature. Some snapshots of the system's output are shown in Fig 2.9.

In the first stage of the experiment, the system is presented with visual stimuli of the digits 1 to 9, in this order. The images at the top of the plot shows the input to the receptive field at the time of the vertical dotted lines. Each number is presented for five seconds, followed by a pause of three seconds, in which no input is presented. In Fig 2.10 the learning phase is marked by a gray shaded background. Next, two test sequences are presented to the DVS: The *Test 1* sequence is composed of the random sequence "1 3 5 7 9 2 4 6 8", using the same presentation and pause times as in the learning phase. The *Test 2* sequence is composed of another random sequence "9 8 7 6 5 4 3 2 1", this time without pauses between digit presentations (which still last for five seconds). These sequences are represented as the ground truth for the experiment by blue horizontal lines in Fig 2.10. For clarity, we re-ordered the ESNs such that the ESN index corresponds to the digit it represents. Successful learning means that the blue lines should align as much as possible with the red dots, representing the output of the WTA network. Occasional deviations are due to noise.

Fig 2.10 shows that each ESN manages to learn complex features, and reliably recognizes them when the respective feature is presented again. This was achieved with raw, noisy DVS inputs, and fully random jitter of the digits during presentation. The experiment shows that complex features can be extracted and recognized also in the absence of characteristic spatiotemporal structure in input patterns.

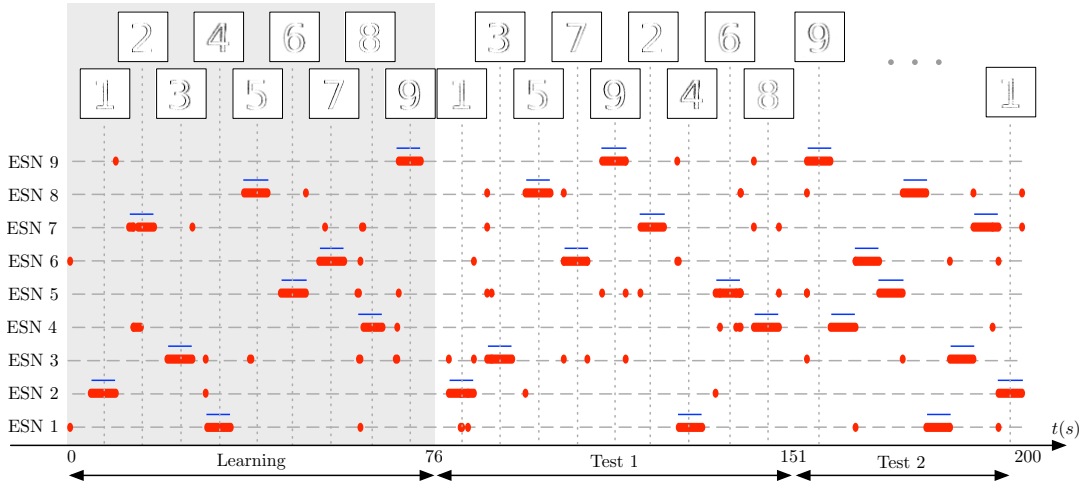


Figure 2.10: Learning complex features from DVS inputs. The DVS records digits from 1 to 9, each animated with random jitter simulating the effect of microsaccades in the biological eye. Blue horizontal lines show the ground truth, indicating when each number was present in the input, and thus when a specific ESN should fire. The corresponding pattern presented to the camera at that time is represented on top of the figure. Red dots show time points where each ESN is selected by the WTA network. The gray shaded area marks the learning phase, in which every digit from 1 to 9 was presented once to the system. Subsequently, two series of testsare shown: First, in the period marked as *Test 1*, the 9 digits are presented in random order, but with short pauses between stimulus presentations. Then, in the period marked as *Test 2*, the digits are presented again in a different random order, but without pauses in between. The results show that also for complex features like digits, every ESN can learn to specialize and represent a distinct feature.

2.4 Discussion

This article presents a new architecture for extracting spatiotemporal visual features from the signal of an asynchronous event-based silicon retina. The spatiotemporal signal feeds into the system through a layer of Echo-State Networks, which compute predictions of future inputs. An unsupervised learning process leads to specialization of ESNs to different features via WTA competition, which selects only the best predictors of the present input pattern for training. Whenever an already learned pattern is presented again, the system can efficiently and reliably detect it. Experimental results confirm the suitability of the feature extraction method for a variety of input patterns. The spatiotemporal feature extraction leads to robust and reproducable detection, which is a key requirement for its use in higher-level visual recognition and classification. A central characteristic of the presented technique, in contrast to conventional computer vision methods, is that it does not depend on the concept of representing visual inputs as whole image frames. Instead, the method works efficiently on event-based sparse and asynchronous input streams, which maintain the temporal dynamics of the scene due to the highly precise asynchronous time sampling ability of the silicon retina. Thus, also the extracted spatiotemporal features contain richer dynamic information, in addition to recognizing spatial characteristics.

Central to the definition of spatiotemporal features in our architecture is the presence of multiple models for prediction, which compete already during learning, such that specialization can occur. Similar concepts are used by various well known machine learning frameworks, most notably the mixture-of-experts architecture [77, 78, 79], in which a gating function creates a soft division of the input space for multiple local "expert" models. The output of the whole network is then a combination of the expert predictions, weighted according to their responsability for the present input. These architectures have been extended in brain-inspired architectures for reinforcement learning and control [80, 81, 82], where multiple forward models and controllers are learned simultaneously, and the prediction performance of the forward model determines the selection of the most appropriate local controller. Mixture-of-experts architectures are closely related to learning mixture models with the EM algorithm [83, 78], where the E-step computes a soft assignment of data points to models. [84] and [85] have proven that this can be implemented in spiking neural networks, using a soft WTA circuit to compute the E-step, and an STDP learning rule to implement the M-step. Compared to these related architectures, our new model advances in three important aspects: Firstly, whereas EM and mixture-of-experts address static input distributions, we here extend this to multiple feature predictors for spatiotemporal sequences. Secondly, our architecture allows online learning of independent features, which contrasts with batch methods like PCA or ICA that operate on the full dataset after its collection. Thirdly, our neural network architecture is specifically designed to work with spiking inputs and for implementation with spiking neurons, thus maintaining the precise dynamics of event-based vision sensors. Other spiking neural network architectures for processing DVS inputs such as spiking ConvNets [86, 87], and spiking Deep-belief networks [88] do not explictly model the dynamics of the features extracted within the networks, but instead rely on different conversion mechanisms from analog to spiking neural networks, without taking sensor dynamics into account. The features they extract are thus characterizing a current snapshot of the input, and do not take its future trajectory into account like the ESN predictors of the presented model, but nevertheless are very useful for fast recognition. This is also true for approaches that directly classify spatiotemporal spike patterns, see e.g. [89, 90]. Spiking network models that represent spatiotemporal dynamics by emulating Hidden Markov Models have recently been introduced [91, 92]. Compared to our approach, these networks do not directly learn dynamic input features, but rather identify hidden states to determine the position within longer sequences.

The combination of visual sensing with bio-inspired artificial retinas and event-based visual feature extraction, as presented in this article, opens new perspectives for apprehending the mechanisms of visual information encoding in the brain. It is clear that the traditional views of visually selective neurons as static image filters for receptive fields, e.g. as Gabor-like orientation filters, which are central to many classical vision models like HMAX or Neocognitron [93, 55], fails to explain how these neurons deal with the highly dynamic and sparse spike inputs from biological retinas. In the presented approach, features are naturally learned and adapted to the task. In Fig. 2.8 it was shown that if the number of available ESNs exceeds the number of features necessary to describe a scene, only the minimum necessary number of networks are trained. This has the desireable effect that whenever a new scene with new features is encountered, the previously unused ESNs can be trained to predict novel stimulus features. This behavior has several benefits: firstly, the number of ESNs does not have to be precisely tuned, but can be set to the highest acceptable number, and only the minimum number of networks is actually recruited and trained as feature detectors by the system. Alternatively, one could employ a different strategy in which new

networks are recruited to the pool, whenever all current ESNs have specialized on features. Secondly, training of feature detectors works completely unsupervised, so no higher-level controller is needed to identify what the elementary features for a scene should be. Although the precesence of a supervisor is not necessary, having such information available would still be beneficial. For instance, another processing layer could use the outputs of the WTA to control the survival of each network. If such processing layer determines that a particular network does not provide enough interesting information, the supervisor could decide to reset and release the associated ESN, so that it can detect more relevant features.

The presented method has great potential for use in event-based vision applications, such as fluid and high-speed recognition of objects and sequences, e.g.in object and gesture recognition [88, 94], or for high-speed robotics [38, 95].

The presented architecture is almost entirely based on computation with spikes. Inputs come in the form of AER events from DVS silicon retinas, providing an event-based representation of the visual scene. The WTA circuit for choosing between feature extractors is also working with spikes, and produces spike outputs, which indicate the identity of the detected feature. The only component of the system which does not entirely use spikes is the layer of ESNs that predict the visual input, but this restriction could be lifted by replacing ESNs with their spiking counterparts, called Liquid State Machines (LSMs) [96], which are computationally at least equivalent to ESNs [97, 98]. The reasons why we have chosen to use ESNs for this proof-of-principle study are the added difficulty of tuning LSMs, due to the larger number of free parameters for spiking neuron models, delays, or time constants, in addition to the higher computational complexity involved in the simulation of spiking neural networks on conventional machines, which makes it hard to simulate multiple LSMs in real-time. Overall, we expect the improvement due to using fully spike-based feature detectors and predictors to be rather minor, since the ESNs can be efficiently simulated at time steps of 1ms, which is also the time interval at which the silicon retina is sending events through the USB bus. However, a fully spike-based architecture does have great advantages in terms of efficiency and real-time executing if it can be implemented entirely on configurable neuromorphic platforms with online learning capabilities [20, 99, 100], which is the topic of ongoing research.

Chapter 3

HOTS: A Hierarchy Of event-based Time-Surfaces for pattern recognition

This chapter describes novel event-based spatio-temporal features called time-surfaces and how they can be used to create a hierarchical event-based pattern recognition architecture. Unlike existing hierarchical architectures for pattern recognition, the presented model relies on a time oriented approach to extract spatio-temporal features from the asynchronously acquired dynamics of a visual scene. Similarly to cortical structures, subsequent layers in our hierarchy extract increasingly abstract features using increasingly large spatio-temporal windows. The central concept is to use the rich temporal information provided by events to create contexts in the form of time-surfaces which represent the recent temporal activity within a local spatial neighborhood.

Contents

3.1 Introduction

Feature selection for object recognition is a fundamental problem in the study of visual processing. The open issue is always to determine how features of an image should be extracted and characterized. In traditional computer vision, visual features are typically defined as a function of the static luminance information within a local spatial neighborhood of an image [65][101]. Different feature types differ only in the function they apply to the image. The temporal content of features has rarely been explored or tackled, mainly due to the three underlying hypotheses on which machine vision is based. The first hypothesis is that scenes are observed using a stroboscopic acquisition which produces a collection of static images (frames). Images are currently at the core of the whole field of artificial vision. So far, everything that has been developed has been designed to acquire, operate on, and display frames. A major drawback of frame-based acquisition is that it acquires information in a way that is independent of the dynamics of the underlying scene [63]. Scene illumination is measured at unnatural fixed time periods (frame-rate), resulting in acquisition of huge amounts of redundant data because most pixels will not change from one frame to the next. Massive redundancy in the acquired data is what allows video compression algorithms to achieve such impressive compression ratios (often around 50:1 [102]). However, before compression, this redundant data is still unnecessarily sampled, digitized, and transmitted, inducing a waste of resources, before expending even more resources to implement compression. This process sets important limitations on artificial perception that might one day be surmounted by using faster and more powerful computing devices (e.g. GPUs, clusters, etc.), but always at the cost of increasing power consumption. Nevertheless, the lack of dynamic content and the acquisition of both relevant and non-relevant data will always be the fundamental limit of images.

The second hypothesis of machine vision is that absolute pixel illumination (gray levels or colors) is the main source of information. However illumination is not an invariant property of a scene [65]. Most current algorithms fail to operate in uncontrolled lighting conditions. The ability to accurately measure luminance is also limited by the low dynamic range of conventional cameras [103].

The third hypothesis is that real-time operation implies a minimum of 24 images per second (the frame rate of common video formats [102]). There is currently a widespread belief in the field of artificial vision that high visual acquisition rates are only useful for cases where a fast changing stimulus must be observed. It is true that sensations of dynamic motion can be observed at 24 fps. However, it has been recently shown that biological retinas operate at temporal precision of 1kHz (see [104]) because that is where most of the information of everyday scenes are [105]. If conventional scenes are processed at low temporal acquisition rates (30-60Hz), it has been shown that there is a loss of 75% of valuable information leading to a poor separability between classes of objects [105]. Currently it is computationally and energetically expensive to process visual input in real time using conventional cameras at above 100Hz. This because the amount of data which must be processed grows linearly with the frame rate, while the amount of information captured only grows sublinearly [105].

However, the field of Neuromorphic Engineering [3] has been developing bio-inspired event-driven, time-based vision sensors which operate on a very different principle [5]. Instead of capturing static images of the scene, these sensors record pixel intensity changes with high temporal precision. This high temporal precision provides information about scene dynamics which can aid in recognition and increase class separability [105]. In the last decade these sensors have matured to a point where they are now commercially available and can be operated by laymen. Event-

driven time-based vision sensors promise to allow for power efficient low latency visual sensing in real world moving scenes, which has the potential for major impact in robotics, as well as mobile and wearable devices.

Event-driven time-based vision sensors provide data in the Address Event Representation (AER) format [30] which differs significantly from frames, and therefore conventional Machine Vision algorithms cannot be directly applied. For the task of object recognition, accuracy using event-driven time-based vision sensors still lags behind traditional approaches. Previous notable works on object recognition using event-driven time-based vision sensors include the Convolution AER Vision Architecture for Real-time (CAVIAR) project [106] which recognizes and tracks circles of different sizes using a hierarchical spiking network running on custom hardware. Later work progressed to differentiation between different shapes (circle, square, triangle) using an HMAX inspired algorithm also on custom silicon hardware [107]. Targeting more complex shapes, Perez-Carrasco *et al.* introduced a card pip recognition task which has been tackled in real-time using FPGAs running different hierarchical spiking models inspired by Convolutional Neural Networks (CNNs) [108] and HFirst [109].

Inspired by the popularity of the MNIST database (Mixed National Institute of Standards and Technology database) [101] in traditional machine vision, there has been a recent focus on character recognition using event-driven time-based vision sensors, which has been tackled using CNNs [108], Hierarchical like models [109], and Deep Belief Networks (DBNs) [110]. Perez Carrasco *et al.* showed how recent advances in training frame based CNNs can be leveraged by converting frame based CNNs to spiking CNNs for recognition [108]. Moving past shape and character recognition, similar hierarchical models have been developed for application to human posture detection [111]. Frame-based CNNs have also been applied for discriminating between vehicles and pedestrians in a traffic scene, and recognizing household objects [112].

This work serves to advance the state of the art for performing recognition using event-driven time-based vision sensors. To provide comparison to previous works, we tackle the previously published card pip dataset [108] and character recognition tasks [109], achieving near perfect accuracy on both. As a first step towards performing human user recognition using event-driven time-based vision sensors, we introduce a new, more challenging, facial recognition task on which we achieve 79% accuracy, providing room for improvement in future work.

This chapter begins with an introduction to the operation of event-driven time-based vision sensors in Section 3.2, followed by a description of the hierachical time-surface feature extraction technique in Section 3.3. We then describe performance of the technique in Section 3.4, before wrapping up with a discussion of results and a conclusion in Section 3.5 and Section 3.6.

3.2 Event-driven time-based vision sensors

In this work, we use event-driven time-based vision sensors as described in Section 1.2. The novel features we propose are thus designed to take advantage of the high temporal resolution data representation provided by event-based cameras, which is not provided by frame-based sensors.

We use datasets recorded by different event-based sensors [113, 26] and previously used in other publications [108, 109]. Due to the high accuracy we achieve on these dataset, we introduce a new set of recordings acquired using the Asynchronous Time-based Image Sensor (ATIS) [12].

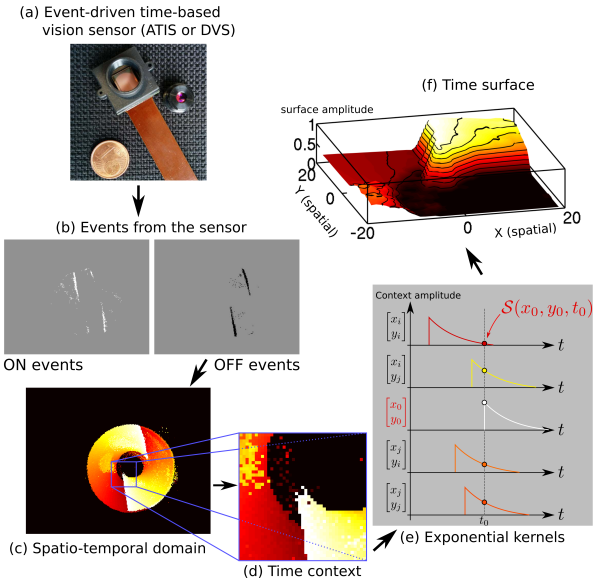


Figure 3.1: Definition of a time-surface from the spatio-temporal cloud of events. A time-surface describes the recent time history of events in the spatial neighborhood of an event. This figure shows how the time-surface for an event happening at pixel $\mathbf{x_0} = [x_0, y_0]^T$ at time t_0 is computed. The event-driven time-based vision sensor (a) is filming a scene and outputs events shown in (b) where ON events are represented on the left hand picture and OFF events on the right hand one. For clarity, we continue by only showing values associated to OFF events. When an OFF event $ev_i = [\mathbf{x_0}, t_i, -1]$ arrives, we consider the times of most recent OFF events in the spatial neighborhood (c) where brighter pixels represent more recent events. Extracting a spatial receptive field allows to build the event-context $\mathcal{T}_i(\mathbf{x}, p)$ (d) associated with that event. Exponential decay kernels are then applyed to the obtained values (e) and their values at t_i constitute the time-surface itself. (f) shows these values as a surface. This representation will be used in the following figures and the label of the axes will be removed for better clarity.

3.3 Model Description

In this section we describe the construction of our architecture for object recognition. We begin by formally defining time-surfaces (Section 3.3.1), and how time-surface prototypes can be learnt from input data (Section 3.3.2). Then we show how these time-surface prototypes can be arranged to form a hierarchical model (Section 3.3.3). Finally, in Section 3.3.4 we describe how classification is performed on the model output.

3.3.1 Time-surface

The process of building a time-surface from the output of an event-driven time-based vision sensor is illustrated in Fig. 3.1 and described hereafter.

Consider a stream of visual events (Fig. 3.1(b)) which can be mathematically defined as

$$ev_i = [\mathbf{x_i}, t_i, p_i]^T, \quad i \in \mathbb{N} \tag{3.1}$$

where ev_i is the i^{th} event and consists of a location ($\mathbf{x_i} = [x_i, y_i]^T$), time ($t_i$) and polarity ($p_i$), with $p_i \in \{-1, 1\}$, where -1 and 1 represent OFF and ON events respectively. When an object (or the camera) moves, the pixels asynchronously generate events which form a spatio-temporal point cloud representing the object's spatial distribution and dynamical behavior. Fig. 3.1(b) shows such events generated by an object rotating in front of the sensor ((Fig. 3.1(a)) where ON and OFF events are represented respectively by white and black dots.

Because the structure of this point cloud contains information about the object and its movement, we introduce the time-surface $\mathcal{S}_i$ of the i^{th} event ev_i to keep track of the activity surrounding the spatial location $\mathbf{x_i}$ just before time t_i. We can then define $\mathcal{T}_i(\mathbf{u}, p)$ a time-context around an incoming event ev_i as the array of most recent events times at t_i for the pixels in the $(2R+1) \times (2R+1)$ square neighborhood centered at $\mathbf{x_i} = [x_i, y_i]^T$ as:

$$\mathcal{T}_i(\mathbf{u}, p) = \max_{j \leq i} \left\{ t_j \,|\, |\mathbf{x_j} = (\mathbf{x_i} + \mathbf{u}),\, p_j = p \right\}, \tag{3.2}$$

where $\mathbf{u} = [u_x, u_y]^T$ is such that $u_x \in \{-R, \ldots, R\}$, $u_y \in \{-R, \ldots, R\}$ and $p \in \{-1, 1\}$ $\mathcal{T}_i(\mathbf{x}, p)$ is shown in Fig. 3.1(d) where intensity is coding for time values: bright pixels show recent activity whereas dark pixels received events further away in the past (only time values corresponding to OFF events are represented in the figure for clarity).

Let $\mathcal{S}_i(\mathbf{u}, p)$, be the time-surface around an event ev_i (shown in Fig. 3.1(e)), it is defined by applying an exponential decay kernel with time constant τ on the values of $\mathcal{T}_i(\mathbf{u}, p)$.

$$\mathcal{S}_i(\mathbf{u}, p) = e^{-(t_i - \mathcal{T}_i(\mathbf{u}, p))/\tau}. \tag{3.3}$$

$\mathcal{S}_i$ provides a dynamic spatiotemporal context around an event, the exponential decay expands the activity of passed events and provides information about the history of the activity in the neighborhood. The resulting surface $\mathcal{S}_i(\mathbf{u}, p)$ is shown in Fig. 3.1(f) for the OFF events represented all along Fig. 3.1. In the following sections $\mathcal{S}_i(\mathbf{u}, p)$ will be referred to directly as $\mathcal{S}_i$ to simplify notations. In the figures it will be represented as a surface showing the values of each of its element at their corresponding spatial positions.
Fig. 3.2 shows examples of time surfaces for simple moving edges. One can see, that a time surface is composed of two halves corresponding to the two polarity of incoming events. The first half has positive values, showing points corresponding to the ON events ($p = 1$) and the second half has negative values showing points corresponding to the OFF events ($p = -1$).

3.3.2 Learning Time-surface prototypes

Time-surface prototypes take the form of time-surfaces themselves, but whereas each incoming event will have a different surface, the time-surface of each prototype remains constant (except during an initial learning phase). Time-surface prototypes are the set of elementary surfaces that are encountered in the observed scenes. The process of learning a time-surface for each prototype is described below, it relies on an incremental clustering process [114].

When an input event arrives at a bank of time-surface prototypes, the time-surface associated to the incoming event is calculated and compared to the time-surface of each prototype. The prototype with the time-surface most closely matching the surface of the input event will then generate an output event. We begin with a set of N inital time-surface prototypes, $\mathbf{C}_n$, $n \in [\![1, N]\!]$,

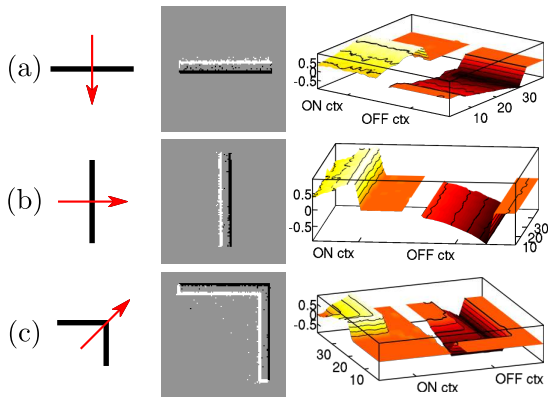


Figure 3.2: Examples of some time-surfaces for simple movements of objects. First column shows a representation of the stimulus. The second column shows corresponding data from the ATIS sensor where white dots are ON events and black dots are OFF events. The third column shows the time-surface obtained from these events: the first, positive, half is obtained from the ON events and the second, negative, half is obtained from the OFF events. (a) A horizontal bar moving downwards. (b) A vertical bar moving rightward. (c) Corner moving to the top-right.

where $\mathbf{C}_n$ takes the same form as $\mathcal{S}_i$ in (3.3). For initialization we simply use the first N time-surfaces as our initial values for the N prototypes. More formally:

$$\mathbf{C}_n = \mathcal{S}_n \qquad n \in [\![1, N]\!] \tag{3.4}$$

We then implement learning using the online clustering algorithm described in [114].

For each input event, ev_i, we calculate the time-surface, $\mathcal{S}_i$, and find $\mathbf{C}_k$, the time-surface prototype closest to $\mathcal{S}_i$ according to the Euclidian distance. $\mathbf{C}_k$ is then updated using:

$$\mathbf{C}_k \leftarrow \mathbf{C}_k + \alpha(\mathcal{S}_i - \beta\mathbf{C}_k) \tag{3.5}$$

with

$$\beta = \cos(\mathbf{C}_k, \mathcal{S}_i) = \frac{\mathbf{C}_k \cdot \mathcal{S}_i}{||\mathbf{C}_k|| \cdot ||\mathcal{S}_i||} \tag{3.6}$$

$$\alpha = \frac{0.01}{1 + \frac{p_k}{20000}} \tag{3.7}$$

where p_k is the number of time-surfaces which have already been assigned to $\mathbf{C}_k$.

The full clustering process is summarized in Algorithm 2.

After this learning process, each time-surface can be associated to a particular prototype $\mathbf{C}_k$. In this manner, the stream of input events is transformed into a stream of prototype activations:

$$feat_i = [x_i, y_i, t_i, k_i]^T \tag{3.8}$$

where k_i is the index of the cluster center $\mathbf{C}_{k_i}$.

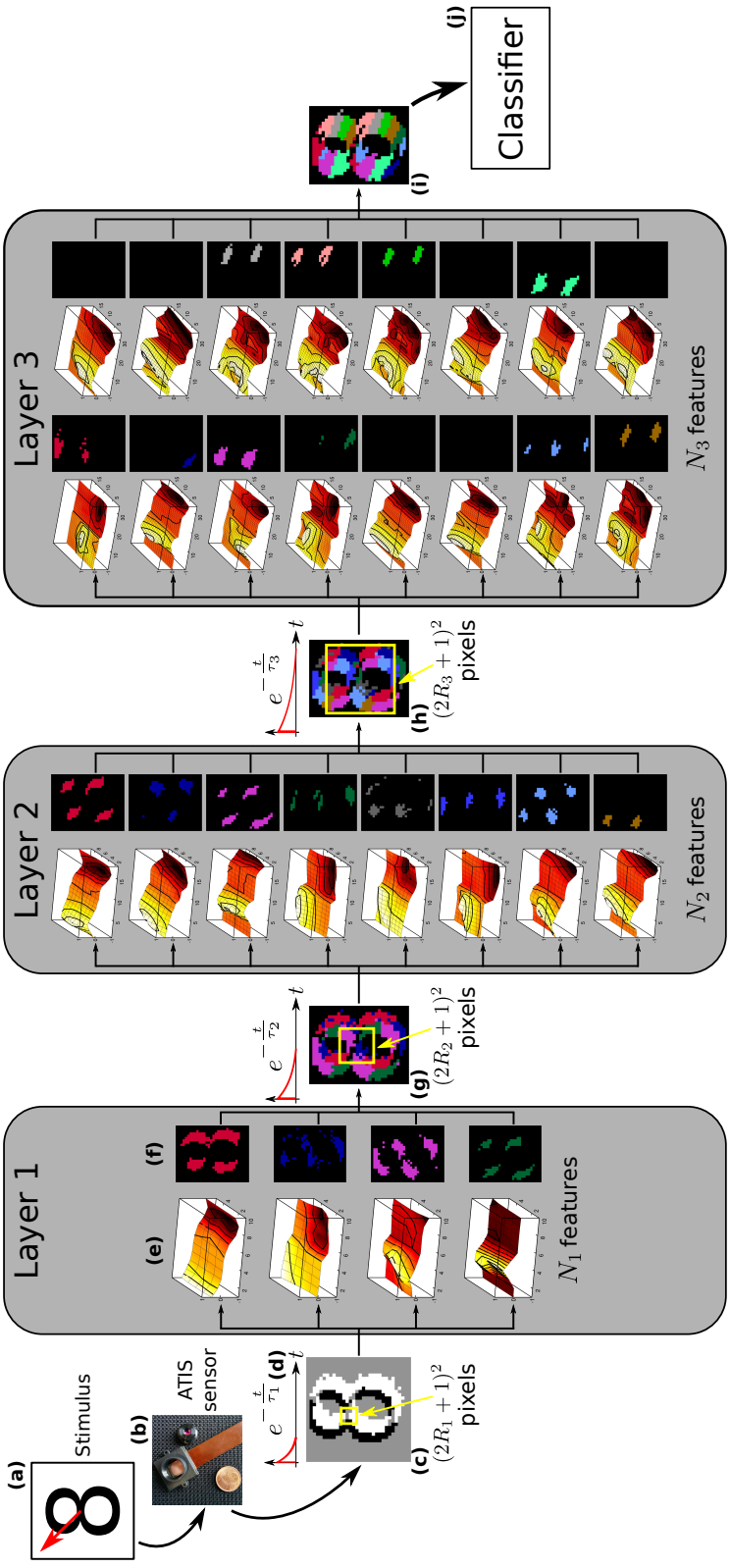


Figure 3.3: View of the proposed hierarchical model. From left to right, a moving digit (a) is presented to the ATIS camera (b) which produces ON and OFF events (c) which are fed into Layer 1. The events are convolved with exponential kernels (d) to build event contexts from spatial receptive field of sidelength $(2R_1+1)$. These contexts are clustered into N_1 features (e). When a feature is matched, it produces an event (f). Events from the N_1 features are merged into the output of the layer (g). Each layer k (gray boxes) takes input from its previous layer and feeds the next by reproducing steps (d)-(g). The output of Layer k is presented between Layer k and $k+1$ ((g),(h),(i)). To compute event contexts, each layer considers a receptive field of sidelength $(2R_k+1)$ around each pixel. The event contexts are compared to the different features (represented as surfaces in the gray boxes) and the closest one is matched. The images next to each features show the activation of their associated features in each layer. These activations are merged to obtain the output of the layer. The output (i) of the last layer is then fed to the classifier (j) which will recognize the object.

Algorithm 2 Online clustering of time-surfaces

Ensure: N cluster centers $\mathbf{C}_n$, $n \in [\![1,N]\!]$
  Use the first N events' time-surfaces as initial values for $\mathbf{C}_n$, $n \in [\![1,N]\!]$
  Initialize $p_n \leftarrow 1$, $n \in [\![1,N]\!]$
  for every incoming event ev_i **do**
    Compute time-surface $\mathcal{S}_i$
    Find closest cluster center $\mathbf{C}_k$
    $\alpha \leftarrow 0.01/(1+p_k/20000)$
    $\beta \leftarrow \mathbf{C}_k \cdot \mathcal{S}/(||\mathbf{C}_k|| \cdot ||\mathcal{S}||)$
    $\mathbf{C}_k \leftarrow \mathbf{C}_k + \alpha(\mathcal{S} - \beta\mathbf{C}_k)$
    $p_k \leftarrow p_k + 1$
  end for

3.3.3 Creating a Hierarchical Model

Fig. 3.3 illustrates the hierarchical model we introduce in this work. Steps (a) to (g) sum up the process described in the previous sections. As shown in Fig. 3.3, a moving digit (a) is presented to the ATIS camera (b) which produces ON and OFF events (c). Time-surfaces are built by convolving them with an exponential kernel of time constant τ_1 (d) and considering spatial receptive fields of sidelength $(2R_1+1)$. These time-surfaces are then clustered into N_1 prototypes represented as surfaces (e) in the Layer 1 box. When a cluster center is matched, an event is produced, resulting in the activations shown in (f). These events are merged to form the output of Layer 1 (g). One can see that each incoming event from the observed pattern is associated with the most representative prototype surface.
The nature of the output of Layer 1 is exactly the same as its input: Layer 1 outputs timed events. Once a prototype matches the temporal surface around the incoming event it immediately emits an event. Thus, the same steps used in Layer 1 (from (d) to (g)) can be applied in Layer 2. However the emitted event is now representing the temporal activity of a prototype surface, it thus carries more meaning than the initial camera event. The prototype surfaces of Layer 2 represent the temporal signature of the activity of complex features. Layer 2 uses different constants for space-time integration of features (R_2, N_2 and τ_2). The goal is to introduce stability of the perceptual representation and sensitivity to the accumulation of sensory evidence over time. This integration over longer and longer time period will thus be able to accumulate evidence in favor of alternative propositions in a recognition process. When alternatives with a barely discernible difference in their sensory inputs are presented over an extended period of time, longer time and spatial integration scales can accumulate the small differences over time until it becomes eventually possible to discriminate the alternatives through its ever growing output. This accumulation dynamics is at the heart of the HOTS model, the difference between time scales can be substantial and can start from 50ms for Layer 1 to 250ms for Layer 2 to finally reach 1.25 s for Layer 3.
Layer 3 receives input from Layer 2, it is the last layer of the system and it provides the highest level information integration, as shown in Fig. 3.3(i) time-surface prototypes are also larger both spatially and temporally. The output of the temporal activity of Layer 3 can finally be used for object recognition by being fed to a classifier (shown in Fig. 3.3(j)).

As stated above, each layer is then defined by only a few parameters (we add an index l for the lth layer of the system):

- R_l, which defines the size of the time-surface neighborhood
- τ_l, the time constant of the exponential kernel applied to events
- N_l, the number of cluster centers (prototypes) learnt by the clustering algorithm.

To increase the information extracted by each subsequent layer, we make these parameters evolve between subsequent layer. For each layer, we define the parameters K_R, K_τ, K_N so that:

$$R_{l+1} = K_R \cdot R_l \tag{3.9}$$
$$\tau_{l+1} = K_\tau \cdot \tau_l \tag{3.10}$$
$$N_{l+1} = K_N \cdot N_l \tag{3.11}$$

The obtained architecture consists in a Hierarchy Of Time-Surfaces (HOTS) which is building and extracting a set of features (the prototypes from the final layer) out of a stream of input events. The time-surface prototypes will then be called time-surface features in the rest of the chapter.

Fig. 3.2 shows what these features could be for the first layer of the achitecture where its input basis is constituted of only two vectors: ON events and OFF events. The other layers have input bases constituted of more vectors (as many as the number of features extracted by their previous layer), thus we could represent their features by a serie of surfaces each corresponding to one feature of the previous layer. Because this representation is harder to relate to the actual input from the camera activating the feature, we chose to fuse these surface into their corresponding activity of ON and OFF events. The features of every layer of the architecture will then be represented as a set of two surfaces such as in Fig. 3.2, showing an image of the activity of ON and OFF events associated to the feature.

3.3.4 Classification

In this section we describe how the output of Layer 3 can be used as features for object recognition. Training of the recognition algorithm consists of two main steps. In the first step, different stimuli are presented to the model to learn the time-surface prototypes (referred to in the next sections as *features*) computed as described in the previous section. This is the training phase of the algorithm (*model*). In a second step, the same learning stimuli are presented to the trained model and a histogram of the time-surface feature activations in the final layer is built for each object class. This is the training phases referred to as the *classifier*. A similar histogram is built for each test stimulus, it can then be compared to trained histograms to determine which object is present in the scene. The choice of the histogram is to show the robustness of the method, historams of activities as we will show are sufficient to provide reliable recognition scores. More complex classifier could be used specially time oriented ones such as Echo State Networks [115] or reccurent networks [116], these would allow the learning of the temporeal dynamics of activated features. However, as we will show in the experiment section, this is not necessary as the mean activity of features activation is sufficient to achieve high recognition scores.
When an object is presented to the camera between instants t_{start} and t_{end}, the time-surface feature activations form the set:

$$\mathcal{F}(t_{start}, t_{end}) = \{feat_i \,|\, t_i \in [t_{start}, t_{end}]\} \tag{3.12}$$

From this set, it is possible to build a histogram $\mathcal{H}$ counting how many times each feature has been activated, independently of its spatial position. This will constitute the signature of the observed objects.

Figure 3.4: Flipped cards experiment: Pattern database.
The database for this experiment consists of the four suits (spades, hearts, clubs, and diamonds) found in a card deck. They are captured by a sensitive DVS sensor as the cards are flipped in front of it (white dots represent ON events and black dots OFF events).

To estimate the distance between two histograms, we use three different distances. We will refer to the standard distance when we will use the euclidian norm of the difference between two histograms (by looking at histograms as vectors in which the k^{th} coordinate is the number of times feature k was matched):

$$d(\mathcal{H}_1, \mathcal{H}_2) = ||\mathcal{H}_1 - \mathcal{H}_2|| \tag{3.13}$$

We will refer to the normalized distance when we will use the euclidian norm of the difference between two histograms which have each been normalized by the number of generated features:

$$d^N(\mathcal{H}_1, \mathcal{H}_2) = \left|\left| \frac{\mathcal{H}_1}{\text{card}(\mathcal{H}_1)} - \frac{\mathcal{H}_2}{\text{card}(\mathcal{H}_2)} \right|\right| \tag{3.14}$$

where $\text{card}(\mathcal{H}_k)$ is the total number of features counted in $\mathcal{H}_k$. We will also use the Bhattacharyya distance [117] defined as:

$$d^B(\mathcal{H}_1, \mathcal{H}_2) = -\ln \sum_i \sqrt{\frac{\mathcal{H}_1(i)}{\text{card}(\mathcal{H}_1)} \cdot \frac{\mathcal{H}_2(i)}{\text{card}(\mathcal{H}_2)}}. \tag{3.15}$$

Because these histograms are characteristics of the classes to recognize, we will refer to them as *signatures* in the rest of the chapter.

3.4 Testing

The proposed method has been tested on three different tasks. The first consists of recognizing pips on poker cards as they are shuffled in front of the sensor to identify their suit. This task, which will be referred to as the *flipped card deck recognition task* (Section 3.4.1), has already been tackled by [118] and [109]. The second task is a simulated reading task in which characters are recognized as they move across the field of view of the sensor. This task, which will be referred as the *letters & digits recognition task* (Section 3.4.2), has already been tackled by [109]. These first two tasks have been chosen to provide comparison to previously published work. The third task is a face recognition task, which will be referred as the *Face recognition task* (Section 3.4.3). The data used for these different tasks are illustrated in Figs. 3.4, 3.7, and 3.11 respectively.

3.4.1 Flipped card deck recognition task

The first experiment is run on the card dataset provided by Linares-Barranco [118] who captured the data using the sensitive DVS [113]. It represents a set of playing cards which are being flipped

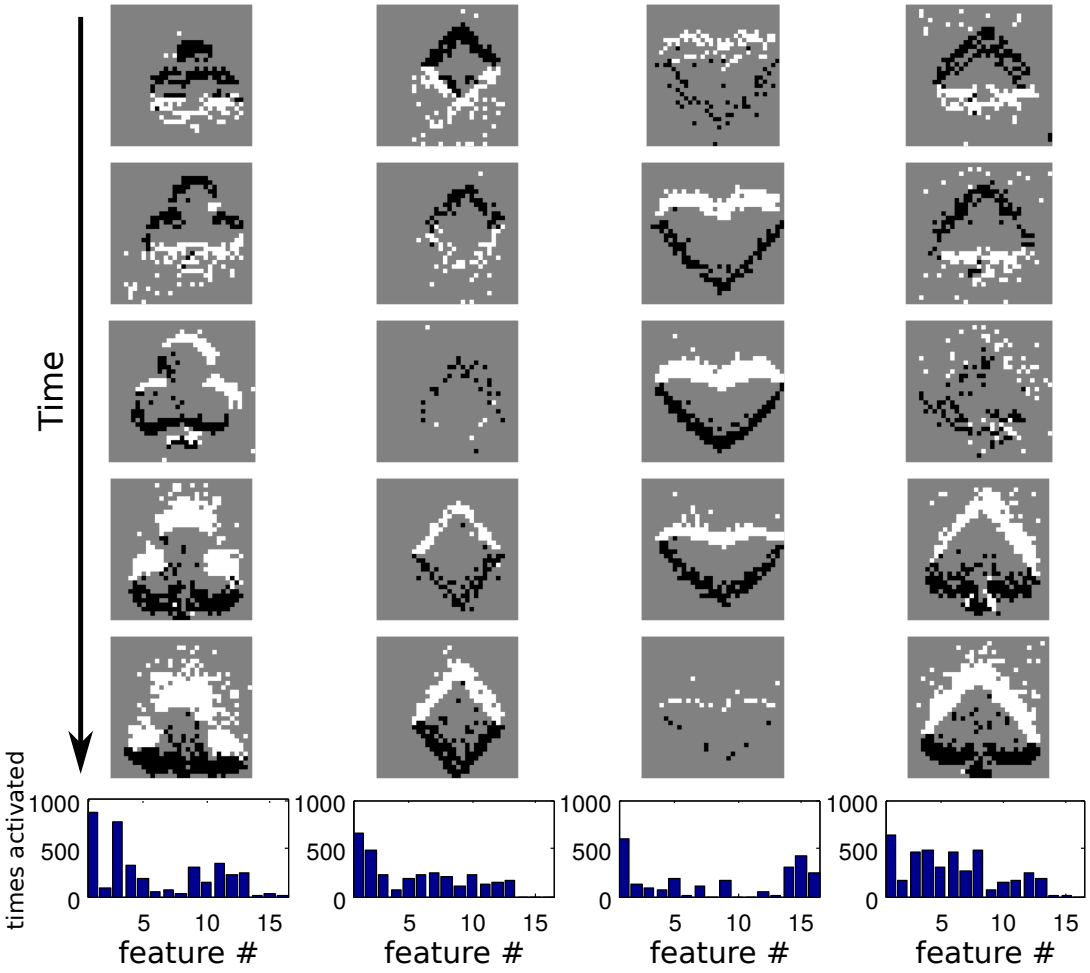


Figure 3.5: Flipped cards experiment: Patterns' signatures.
Histograms of feature activation numbers for the four suits moving in front of the camera. X axis is the index of the feature shown in the supplemental material, Y axis is the number of activations of the feature during the stimulus presentation. Each column corresponds to one suit. The snapshots show how the pips evolve during one particular presentation (each snapshot is taken at a regular time interval). Each pattern outputs a different signature that allows its recognition.

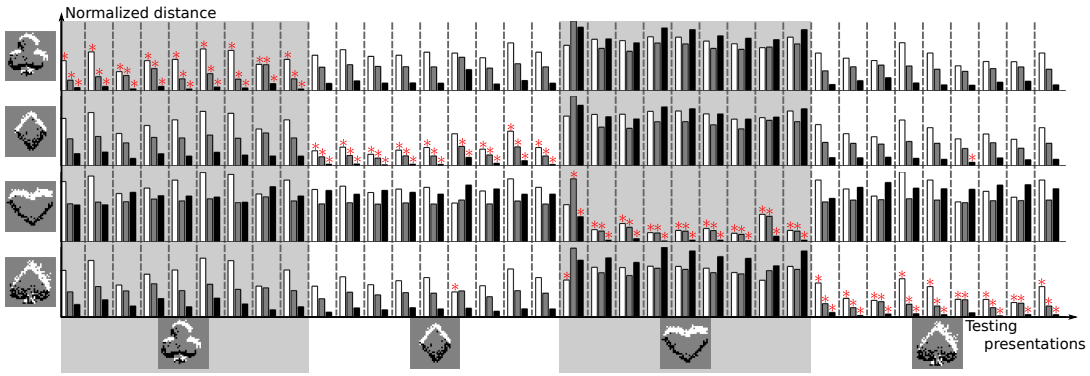


Figure 3.6: Flipped cards experiment: Distance measurements between learning and testing presentations of patterns.
The system is presented with nine different presentations of each learnt pattern. Each line presents data related to a particular trained pattern. Each section in between dashed vertical lines corresponds to the presentation of a test stimulus. Histograms show normalized distances obtained during the experiment so that the recognized objects is the smallest bar in each column (marked with a star). White bars code for standard distance, grey bars for normalized distance and black bars for Bhattacharyya distance. These three distances lead to respective performances of 94%, 100% and 97%.

in front of the sensor (see Fig. 3.4 and Fig. 3.5). The pip representing the suit of the card has been isolated from the recording and the classifier has to determine the suit of the presented card. The data consists of ten presentations of each of the four suits, in the following, one presentation will be used for learning and the other nine for testing. Because the cards are being flipped by hand at high speed during the recording, an important deformation of the symbols occurs.

We use the hierarchical system described in the previous section with 3 layers. The parameters for the first layer are:

- $R_1 = 2$,
- $\tau_1 = 20\,\text{ms}$,
- $N_1 = 4$.

To go from one layer to the next, we use the following parameters:

- $K_R = 2$,
- $K_\tau = 10$,
- $K_N = 2$.

The sensor feeds spiking data into the first layer of the hierarchical model and histograms are built from the third layer's output. All four suits are used to train the model and each layer is trained sequentially. The features extracted in each layer are presented in the supplemental material.

Training stimuli (a single presentation for each suit) are then presented to the system again to train the classifier. Examples of the classifier histograms are shown in Fig. 3.5. The rest of the stimulus examples (nine presentations) for each object are used for testing. The results can be seen in Fig. 3.6. In Fig. 3.6 the model and classifier have both been trained with only one presentation of each of the four suits. The nine other stimulus examples for each suit are used for testing.

The four rows along the vertical axis each show results for a different suit during testing. Each section between dashed lines shows histogram distances for one card flipped in front of the sensor. The different bars encode the histogram distances where white, gray, and black bars code for the standard, normalized, and Bhattacharyya distances respectively. The recognized class is the one with the smallest bar in each column, and is marked with a star. This particular experiment leads to performances of 94%, 100% and 97% with the standard, normalized, and Bhattacharyya distances respectively.

Running some cross validation tests on the data gave us performances of $95\% - 100\%$ with all three distances.

3.4.2 Letters & Digits recognition task

Figure 3.7: Letters & Digits experiment: Pattern database.
The database for this experiment consists of the 26 letters of the roman alphabet and the digits 0 to 9. They are captured by a DVS sensor as the characters are moving in front of it (white dots represent ON events and black dots OFF events).

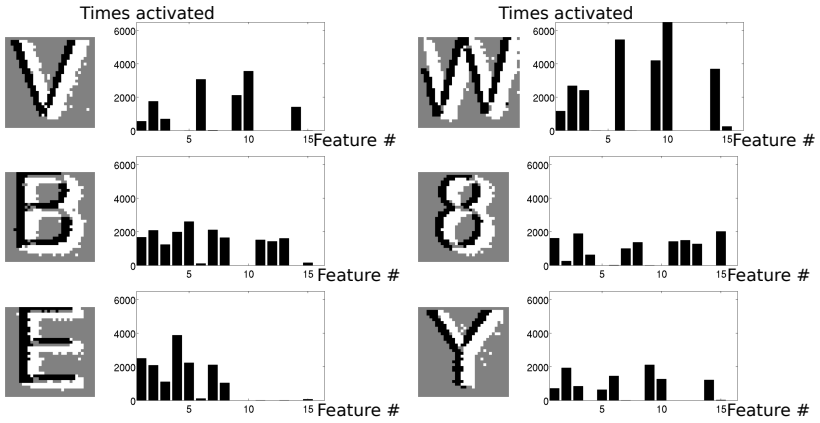


Figure 3.8: Letters & Digits experiment: Pattern signatures for some of the input classes. For each letter and digit the trained histogram used as a signature by the classifier is shown. The snapshot shows an accumulation of events from the sensor (White dots for ON events and black dots for OFF events). The histograms present the signatures: X axis is the index of the feature shown in Fig. C.2, Y axis is the number of activations of the feature during the stimulus presentation. The signatures of all the letters & digits are presented in the supplemental material.

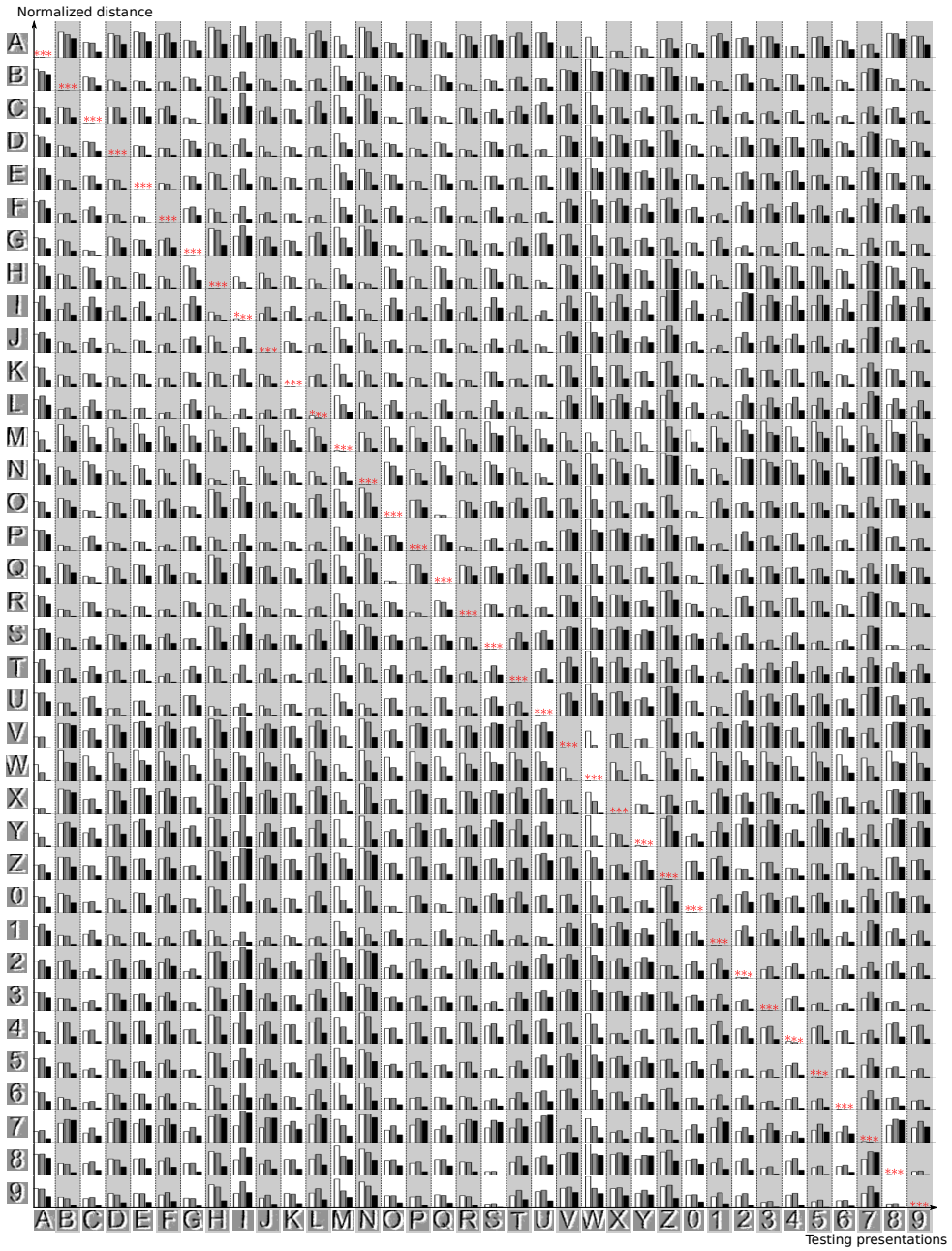


Figure 3.9: Letters & Digits experiment: Distance measurements between learning and testing presentations of patterns.

one presentation of each learnt pattern is shown to the system. Each line presents data related to a particular trained pattern. Each section in between dashed vertical lines corresponds to the presentation of a test stimulus. Histograms show normalized distances obtained during the experiment so that the recognized objects is the smallest bar in each column (marked with a star). White bars code for standard distance, grey bars for normalized distance and black bars for Bhattacharyya distance. These three distances all lead to a 100% recognition rate.

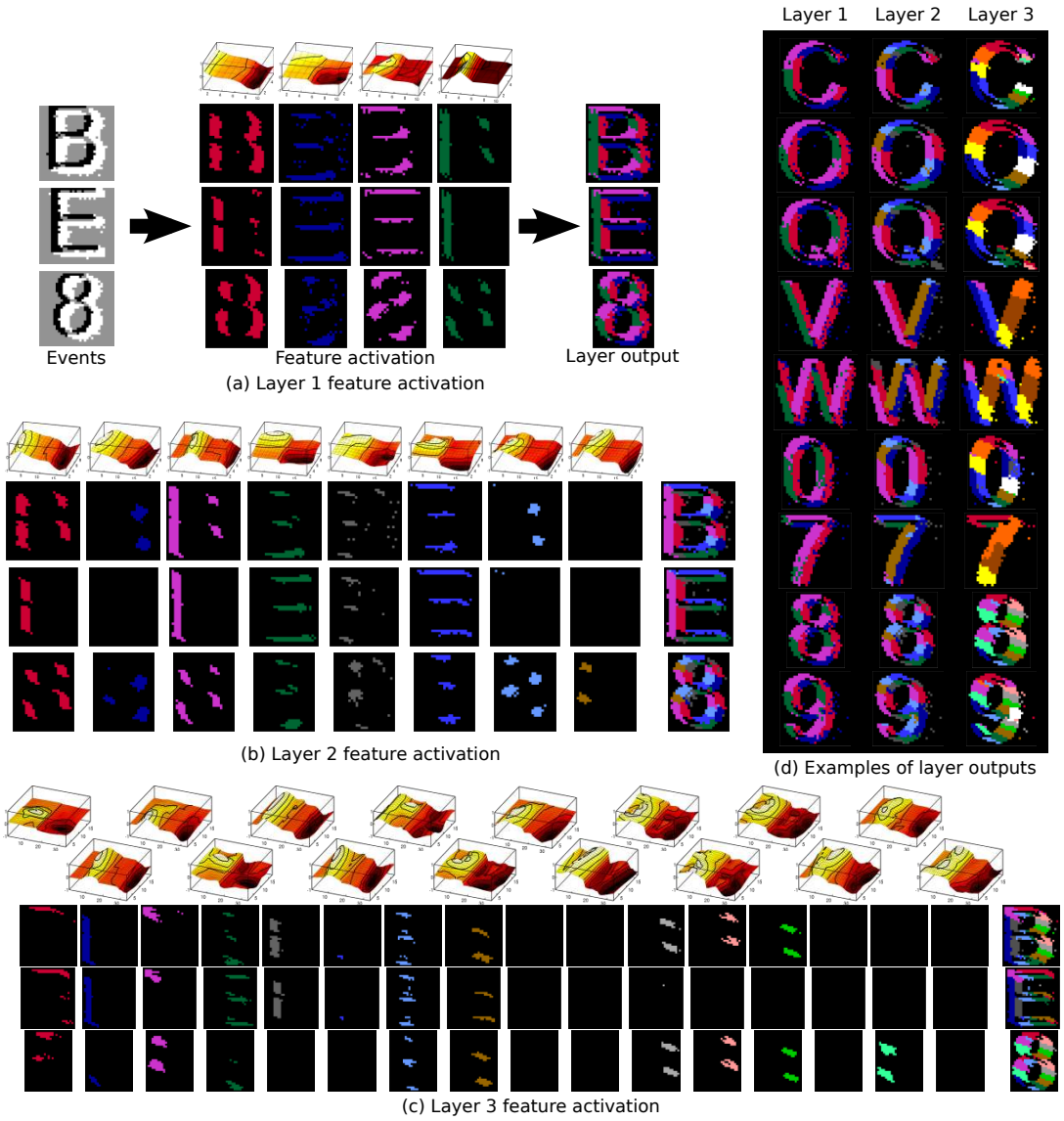


Figure 3.10: Letters & Digits experiment: Activation of features in the different layers. (a), (b) and (c) activation of the different features extracted by each of the three layers for some chosen characters: **E**, **B** and **8**. The last column shows the superposition of the different features (each color corresponds to a particular feature). The other columns show the independent activation of the different features with their associated surface representation. Each of the three line of activations presents a different character.(a) shows the output of the first layer, (b) of the second and (c) of the third. Differences in the information offered by each layer to differentiate between these similar classes can be observed. (d) Examples of outputs of the three layers for a set of recognized objects. We can see how the information becomes more abstract when increasing layers, going from orientation of contours of a given polarity (ON or OFF events) to contours of a given curvature.

The second experiment is run on a dataset provided by Orchard [109]. A DVS camera [26] is presented with the 10 digits from 0 to 9 and the 26 letters from the roman alphabet (see Fig. 3.7) printed on a barrel which was rotated at 40rpm. The goal is to be able to classify correctly these 36 objects. This dataset consists of 2 presentations for each of the 36 patterns.

We use the same hierarchical system described previously with the same parameters as in the previous section.

The camera is feeding data into the first layer and the third layer's output is used by the classifier to recognize the letters and digits. Only the objects **F**, **V**, **O**, **8** and **B** are used to train the model. Each layer is trained sequentially. The trained features are available in the supplemental material. Then, for each digit or character, we use one example to train the classifier. Some examples of the classifier histograms are provided in Fig. 3.8. All the signatures are also available in the supplemental material. One testing presentation is then used for each object to test the classifier. The results can be seen Fig. 3.9. Each row along the vertical axis shows results for a different class during testing. Each section between dashed lines shows histogram distances for one character presented to the sensor. The different bars encode the histogram distances where white, gray, and black bars code for the standard, normalized, and Bhattacharyya distances respectively. The recognized class is the one with the smallest bar in each column, and is marked with a star. Each character is presented once to the sensor in the order we gave to the classes so that perfect results correspond to stars only on the main diagonal. In this experiment, all distances lead to a recognition performance of 100%.

We ran a cross validation test by randomly choosing for each pattern which presentation would be used for learning (both the model and classifier), and the other is then used for testing. Every trial amongst the hundred we ran gave a recognition accuracy of 100%.

This experiment is run on a dataset composed of objects with very distinct spatial characteristics. Because of that, it is the best one to try to interpret the features learnt by the architecture. Fig. 3.10 presents the activation of all three layers' features for three different characters. We can observe on panels (a), (b) and (c) how the information available in different layers allow us to discriminate between three similar characters: **E**, **B** and **8**. Each column shows the response of a given feature (its associated surface representation is shown at the top) when the characters are presented (each line), with the last column showing all these data at once. We can see the difference in activation of the features corresponding to the differences in the input stimuli.

Panel (d) shows the accumulated feature activations for a set of objects used in this task. We can clearly see that the information encoded by each feature is becoming more and more abstract as we go from one layer to the next. In the second layer, features seem to respond to particular orientation of edges constituted of either ON or OFF events. In the third layer however, it seems that these features were pooled in order to recognize the line drawing the characters with features being tuned to its curvature. We can also see that the feature activations are very reproductible from one character to another containing the same inner shapes.

3.4.3 Face recognition task

The results obtained in the previous sections encouraged us to run the proposed method on more complex data. For our last experiment, we use a dataset consisting of the faces of seven subjects. Each subject is moving his or her head to follow the same visual stimulus tracing out a square path on a computer monitor (see Fig. 3.11). The data is acquired by an ATIS camera [13]. Each subject has 20 presentations in the dataset of which one is used for training and the others for testing.

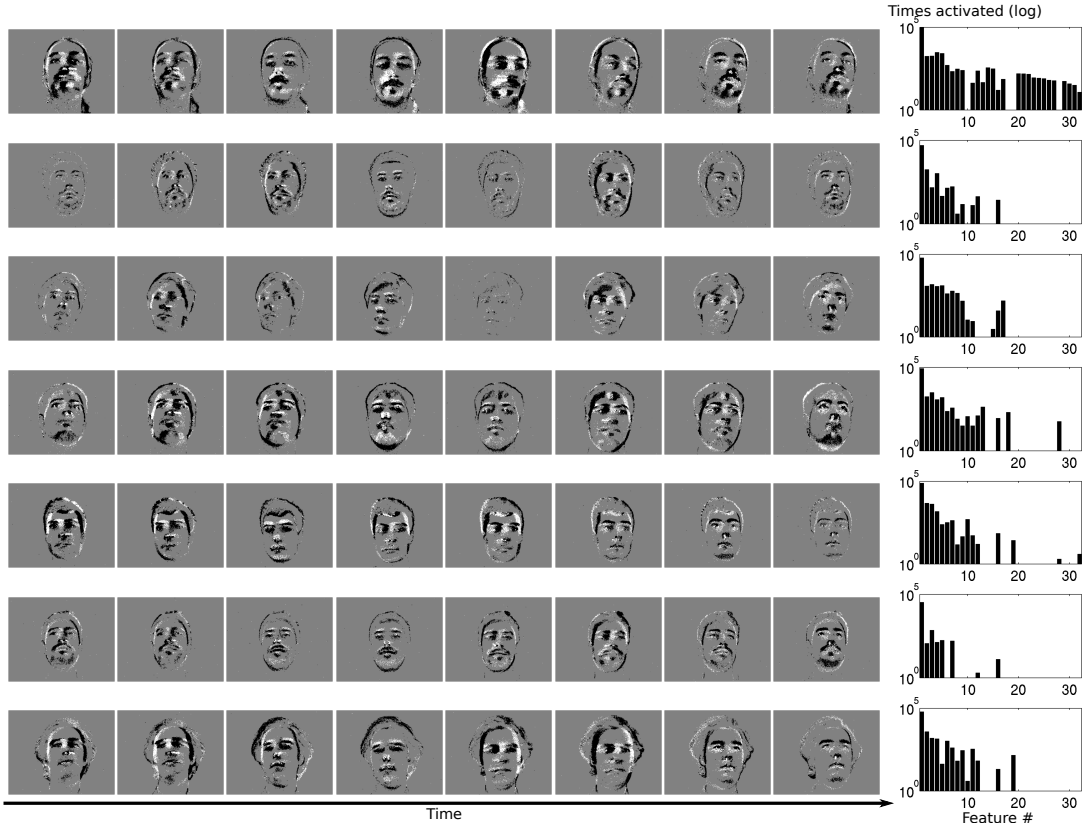


Figure 3.11: Faces recognition experiment: Pattern database and signatures.
The database is constituted of 7 faces moving their gaze by following a dot in a square movement. The snapshots show the obtained stimulus with successive positions in time of the faces (white dots represent ON events and black dots represent OFF events; snapshots are taken every 350 ms.). The last column presents the signature learnt by the classifier for each face using the 32 features of the third layer of the system: X axis is the index of the feature shown in the supplemental material, Y axis is the number of activations of the feature during the stimulus presentation (in logarithmic scaling).

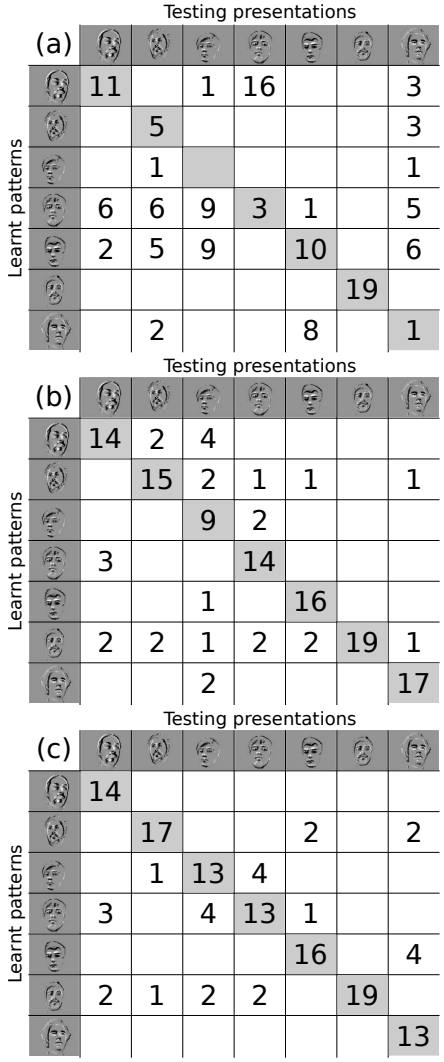


Figure 3.12: Face recognition experiment: Recognition results.
Subfigure show the recognition results for the three different distances used. (a) Standard distance: 37% accuracy. (b) Normalized distance: 78% accuracy. (c) Bhattacharyya distance: 79% accuracy. The system is presented with 19 testing presentations of each learnt class. These 19 presentations are merged into the columns and the numbers are indicating how many matches are obtained for the different learnt patterns. Perferct results would fill the diagonal (gray background cells) with values 19, numbers in other cells correspond to classification errors.

We again use the same hierarchical system described previously with the following parameters:

- $R_1 = 4$,
- $\tau_1 = 50\,\text{ms}$,
- $N_1 = 8$.
- $K_R = 2$,
- $K_\tau = 5$,
- $K_N = 2$.

Because the faces are bigger and more complex objects, we use bigger receptive fields to define the time-surfaces and we set the layers to cluster twice as many features as in the previous experiments. These parameters lead to recognition performances of 37%, 78% and 79% when using the standard, normalized and Bhattacharyya distances respectively.

3.5 Discussion

In this chapter we have described a hierarchical architecture for object recognition using a new type of feature relying on *time-surfaces*. These time-surfaces use the high temporal resolution of event-driven time-based vision sensors to associate a descriptor to every event based on its relative timings to recent activity in its spatial neighborhood. The model then clusters this space to extract features. Successive layers perform this operation again and again, incorporating larger and larger spatial and temporal scales in the process. This allows for features of these successive layers to acquire more information, from bigger spatial receptive fields and from longer timescales.

Every layer also automatically learns its own features from its predecessor's output, removing the need for supervision. The process of training the model is thus completely unsupervised. Supervision only takes place when training the classifier, which inherently requires knowledge of the object class.

Other object recognition methods such as HMAX [60] or HFirst [109] extract orientations as a first step using oriented Gabor filters, thus only looking at the spatial repartition of the input data or events to build features. HFirst does make use of time in computation, but it uses time to encode signal strength of spatial features rather than capturing scene dynamics. In contrast, time-surfaces use both spatial and temporal information to build features which are then not only spatial, but spatio-temporal features. Time-surfaces are, by design, encoding spatial information such as shapes and temporal information such as motion in the feature space. This makes them interesting features for recognition in dynamic environments. More than recognizing moving objects, these features should be able to extract the intrinsic relative inner movements of objects to help the recognition process.

Masquelier *et al.* [119] used STDP to learn features as event patterns. But STDP can only extract the co-activation of different input neurons and thus cannot extract the exact order of activation. Because time-surfaces encodes relative timings, they can distinguish between different orders of activation which is important for real spatio-temporal features. Moreover because time-surfaces are generated around every event, we also do not need to stabilize or track objects for

their recognition and features will be recognized whatever their position in the field of view of the sensor.

The letters & digits experiment demonstrates the generalization strength of the feature extraction process. To recognize the 36 characters constituting the dataset, we only used a subset of 5 characters representative of the whole. When the different layers were learning features, they were only shown the characters **F**, **V**, **O**, **8** and **B**. Then, signatures were obtained for all of the 36 characters of the dataset to train the classifier stage. This still leads to a recognition performance of 100% which is showing us than the system can recognize objects without the need for its features to be specifically trained on those objects.

Recognizing a time-surface essentially consists of performing coincidence detection on input event arriving in a specific spatial and temporal pattern. For instance, this could be implemented by using leaky integrate and fire neurons and delay lines. Thus, the model described in this chapter can be seen as a Spiking Neural Network and could be implemented on neuromorphic hardware for neural simulation. Moreover, every pixel is considering its own receptive field to build its associated time-surface. This can allow most of the processing to be parallelized to be implemented on platforms such as FPGAs or new, highly parallel, computer architectures such as SpiNNaker [120].

Author	Cards dataset	Letters & Digits	Faces
Perez-Carrasco *et al.* [118]	$90.1\% - 91.6\%$	–	–
Orchard *et al.* [109]	$\mathbf{97.5\% \pm 3.5\%}$	$84.9\% \pm 1.9\%$	–
Lagorce *et al.* (Hereby) *Standard distance*	$\mathbf{95\% - 100\%}$	$\mathbf{100\%}$	37%
Lagorce *et al.* (Hereby) *Normalized distance*	$\mathbf{95\% - 100\%}$	$\mathbf{100\%}$	$\mathbf{78\%}$
Lagorce *et al.* (Hereby) *Bhattacharyya distance*	$\mathbf{95\% - 100\%}$	$\mathbf{100\%}$	$\mathbf{79\%}$

Table 3.1: Comparison with state of the art

3.6 Conclusion

We have presented a hierarchical recognition architecture using a new way of representing features in the spatio-temporal output of asynchronous change detection vision sensors. These features are using relative timings of events, enabled by the high temporal precision of these sensors' output, to give contextual information to events, which we have called time-surfaces. The proposed architecture matches or improves the current state of the art on two previously published recognition tasks and results for a third, more difficult task have been presented. This comparison is summed-up in Table 3.1.

Chapter 4

A framework for plasticity implementation on the SpiNNaker neural architecture

Many of the precise biological mechanisms of synaptic plasticity remain elusive, but simulations of neural networks have greatly enhanced our understanding of how specific global functions arise from the massively parallel computation of neurons and local Hebbian or spike-timing dependent plasticity rules. For simulating large portions of neural tissue, this has created an increasingly strong need for large scale simulations of plastic neural networks on special purpose hardware platforms, because synaptic transmissions and updates are badly matched to computing style supported by current architectures. Because of the great diversity of biological plasticity phenomena and the corresponding diversity of models, there is a great need for testing various hypotheses about plasticity before committing to one hardware implementation. Here we present a novel framework for investigating different plasticity approaches on the SpiNNaker distributed digital neural simulation platform. The key innovation of the proposed architecture is to exploit the reconfigurability of the ARM processors inside SpiNNaker, dedicating a subset of them exclusively to process synaptic plasticity updates, while the rest perform the usual neural and synaptic simulations.

Contents

4.1 Introduction

Learning is crucial for the survival of biological organisms, because it allows the development of new skills, memories, and behaviors, in order to adapt to the information acquired from their local environment. Such high-level changes of behavior are the manifestation of an intricate interplay of synaptic plasticity processes, which lasts from early development throughout the adult life, and is taking place simultaneously and continuously in all parts of the nervous system. Although neuroscience has developed an increasingly better insight into the local plasticity mechanisms at specific types of synapses, we still have a poor understanding of the global effects of plasticity that lead to the emergence of our astonishing cognitive capabilities. Clearly, this is one of the great unsolved questions, not only for neuroscience, but with great implications for fields like philosophy, psychology, medicine, and also for engineering disciplines concerned with the development of artificial intelligent systems that can learn from their environment.

Much of our understanding of the functional effects of local plasticity comes from theoretical and simulation studies of simplified learning rules in neural network models. Most influential is the hypothesis of [121], which says that synaptic connections strengthen when two connected neurons have correlated firing activity. This has inspired many classical models for associative memory [122], feature extraction [123], or the development of receptive field properties [124]. Later, the discovery of Spike-timing Dependent Plasticity (STDP) [125, 126] has led to a number of models that have exploited the precise timing properties of spiking neurons for receptive field development [127, 128], temporal coding [129, 130], rate normalization [131, 132], or reward-modulated learning [133, 134, 135, 136]. It has also been realized that there is not one standard model for STDP, but that there is a huge diversity of learning rules in nature, depending on species, receptor, and neuron types [137, 138], the presence or absence of neuromodulators [139, 140], but also on other factors like post-synaptic membrane potential, position on the dendritic arbor, or synaptic weight [141].

The discovery that basic effects can be achieved with local learning rules has had a big influence on the development of larger scale learning models that have mapped methods from machine intelligence onto spiking neural networks. Examples include supervised learning methods for classification of visual (e.g. [142, 143]), or auditory stimuli [144], and unsupervised learning methods like Expectation Maximization [145, 146], Independent Component Analysis [147], or Contrastive Divergence [148]. This has opened up the possibility of using spiking neural networks efficiently for machine learning tasks, using learning algorithms that are more biologically plausible than backpropagation-type algorithms typically used for training artificial neural networks.

The increased interest in spiking neural networks for basic research and engineering applications has created a strong interest for larger, yet computationally efficient simulation platforms for trying out new models and algorithms. Being able to easily and efficiently explore the behavior of different learning models is a very desirable characteristic of a such platform. The major problem for computation with spikes is that it is a resource-intensive task, due to the large number of neurons and synapses involved. Synaptic activity, and specifically synaptic plasticity, which might be triggered by every spike event, is dominating the computing costs in neural simulations [149, 150], partly because the communication and processing of large numbers of small messages (i.e. spikes), is a bad match for current von Neumann architectures. Different strategies to improve the scale and run-time efficiency of neural simulations either rely on supercomputer simulations [151, 152], parallel general-purpose devices such as GPUs [153] and FPGAs [154], or special purpose *neuromorphic* hardware [155]. Each solution involves a trade-off between efficiency,

reconfigurability, scalability and power consumption.

In this context we present a framework for studying arbitrary plasticity models on a parallel, configurable hardware architecture such as SpiNNaker. The SpiNNaker system [156, 120] has been designed as a massively parallel, highly reconfigurable digital platform consisting of multiple ARM cores, which optimally fits the communication requirements for exploring diverse synaptic plasticity models in large-scale neural simulations. Previous implementations of plasticity on SpiNNaker have been limited in their ability to model arbitrary spike- and rate-based learning rules. Here, we present a new approach for implementing arbitrary plasticity models on SpiNNaker, using a dedicated plasticity core that is separated from other cores that process other neural and synaptic events. Specifically we demonstrate the implementation of three synaptic plasticity rules with very different requirements on the trigger events, and on the need to store or access additional variables for computing the magnitude of updates. We show that the same architecture can implement the rate-based BCM rule [124], an implementation of standard STDP based on a model by [157], and a voltage-dependent STDP rule suggested by [142]. We compare the efficiency and correctness of the STDP rule to previous implementations on SpiNNaker, and provide the first implementation of BCM and the learning rule of [142] on this platform. All the experimental results presented in this work come from implementations of learning rules on a 4-chip SpiNNaker board.

The ability to implement different rules with very different requirements, that are either based purely on spike-timing, on the correlation of firing rates, or on additional voltage signals indicates that the framework can be used as a generic way of implementing plasticity in neural simulations. This new architecture therefore provides an efficient way for exploring new network models that make use of synaptic plasticity, including novel rules and combinations of different plasticity rules, and pave the way towards large-scale real-time learning systems.

This article is organized as follows: the next Section introduces different approaches to model learning, from a theoretical and an implementation point of view. Section 4.3 describes the SpiNNaker system, the previous solutions for plasticity on SpiNNaker and our novel approach presented in this work. The flexibility of the framework introduced is demonstrated by the implementation of three different rules, presented in Section 4.4, 4.5 and 4.6: Spike-Timing Dependent Plasticity [129], the rate-based BCM rule [124] and the voltage-dependent variation of the STDP rule [142]. We validate the implementation by replicating classical plasticity experiments, and discuss the performances of each rule in Section 4.7. The chapter is concluded in Section 4.8, which also provides an outlook towards future applications.

4.2 Learning in spiking platforms

The use of parallelization to mitigate the computational costs and difficulties of modeling large plastic networks has been exploited using different tools and strategies. Using many processors in a supercomputer is an important exploratory solution, which can be used to rapidly implement and test learning rules. However, setting up a Message Passing Interface (MPI) mediating the spike communication is a challenging process on a distributed von-Neumann architecture, because the network infrastructure is optimized for large-frame transfers [151, 152] as opposed to small spike packets.

Dedicated neuromorphic [2] systems are natural candidates for emulating parallel neural computation. On these systems, circuits modeling neurons and synapses can be replicated using Very

Large Scale Integration technology (VLSI, [155]). Synapses usually take up the majority of the resources, in terms of computation and chip area. It is also particularly challenging to design plastic hardware synapses. In the FACETS wafer-scale hardware [158], for example, the area of plastic synapses is minimized by separating the accumulator circuit for the spike-timing dependency and a global weight-update controller, which drives the update of multiple synapses [159]. Having a separate plasticity engine makes the update slower, but adds flexibility to the plasticity algorithms that can be implemented. The trade-off in this case relates to the controller frequency update, which evolves slower than the neural dynamics, and the precision of the synapses, limited to 4-bits. Despite these limitations the system is capable of modeling a variety of plasticity models, characterized by different weight dependencies. Also, the synaptic resolution is shown to be not-critical in the simulation of a series of network benchmarks. [160] have introduced a general system where synapses are stored in digital memory with a processor implementing the synaptic update mechanism, while a separate set of ASICs implement the neural integration process. While they demonstrate STDP, more general functions can be implemented using the same scheme.

[142] proposed a learning rule that captures biological properties such as memory preservation and encoding. Furthermore, it is optimized for efficient implementation in a neuromorphic system. The rule is dependent on the post-synaptic neuron membrane potential and recent spiking activity at the time of a pre-spike arrival. Every synapse has internal dynamics, which drives the weight towards a bistable state. Its advantage for VLSI implementations [99, 161, 162] lies in its ability to smooth device mismatch by applying a threshold to the internal state variable, in order to set the synapses to one of two possible states. The bistable representation of memory has the additional advantage of being power efficient. The fact that the rule can be computed when a pre-synaptic spike is received reduces the chip area required by a synapse, and consequently increases the number of synapses that can be modeled. This assumes that the synapses are located on the post-synaptic neuron, and have access to the neural and synaptic state variables when a spike is received. This is the case in the VLSI devices mentioned above and also in SpiNNaker [163]. A review of different neuromorphic approaches and challenges in designing plastic synapses can be found in [164], which discusses power consumption, area requirements, storing techniques, process variation and device mismatch.

Recently, Resistive Random Access Memories, commonly referred as memristors, have raised interest in the neuromorphic community. They are small, power-efficient devices that can be used to store weights and thereby increase the amount of neurons and synapses that can be integrated in a chip. Weight change can be induced by controlling the voltage at the terminals of a memristor, inducing a change in its state and thus modeling a learning rule such as STDP [165] or triplet-based STDP [166]. In [167] memristors are used directly to model synaptic dynamics, using them both for computation and memory storage.

There are also difficulties when implementing synaptic plasticity in general purpose hardware. Regarding GPUs [153], for example, propose a simplified nearest-neighbor pairing scheme with a time-limited STDP window. They continuously accumulate STDP statistics that are then used to update synapses at fixed intervals. In such implementation, increasingly shorter intervals impact performance, lowering the overall spike throughput of the platform. Weight change accumulation is commonly used in other GPU approaches, e.g. in [168], where the synaptic kernel update is applied every second, and in software simulations [169].

The diversity of approaches for studying synaptic plasticity in hardware, indicates a need for general purpose, massively parallel, and reconfigurable computing platforms. Only this will allow fast prototyping of plasticity rules, and their exploration in large scale models, which can in a

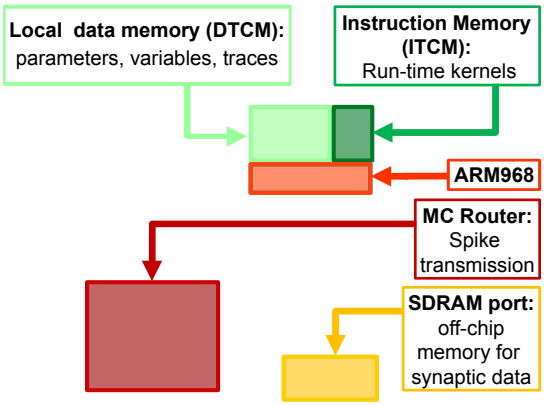


Figure 4.1: High-level view of the SpiNNaker chip, showing: the ARM cores with their Instruction and Data Tightly coupled memory (DTCM and ITCM, 32 and 64 Kbyte respectively) to run applications and locally store data; the Multicast (MC) router responsible of spike transmission; the port to the 128 Mbyte SDRAM off-chip memory, containing the synaptic data.

second stage directly lead to dedicated hardware implementations.

4.3 A novel framework for plasticity implementation on SpiNNaker

SpiNNaker [170, 120] is a digital multi-core, multi-chip architecture designed for the simulation of large neural networks in real time. Each SpiNNaker chip is equipped with a 1Gbit SDRAM and 18 programmable ARM968 cores embedded in a configurable packet-switched asynchronous fabric [171].

The SpiNNaker network infrastructure is designed with spiking communication in mind: every chip contains an on-chip Multicast (MC) router capable of handling very efficiently one-to-many communication of spikes (MC packets). The router links every chip to its six neighbours. Each core has a small local tightly-coupled memory (32 kByte instruction and 64 Kbyte data, ITCM and DTCM respectively). The massive synaptic data required for neural simulations is stored in the shared, off-die SDRAM 128 MByte chip that can be accessed by the cores through DMA requests, for an aggregate read/write bandwidth of 900 MBytes/s [172]. The system is designed to scale up to 60,000 chips for a total of over one million ARM cores. The goal of the system is to simulate 1% of the human brain in real time.

A high level view of the main chip components is presented in Figure 4.1. When simulating neural networks, spikes are delivered and processed by the ARM cores, which update the states of the neurons and synapses. A C-based API is used to program neural kernels [173]. The API offers an accessible interface to the hardware substrate and to real-time event scheduling facilities, and can be used to write applications that are executed in parallel on the machine. The API promotes an event-driven programming model: the neural kernels are loaded into the ARM cores and are used to configure *callbacks* that respond to *events*. A timer event allows the periodic execution of functions, such as neuron state update. A packet event signals the arrival of an MC packet (spike) and can be used to initiate a request to transfer synaptic data from SDRAM. Finally, a memory event indicates that the requested data is available for processing. The neural

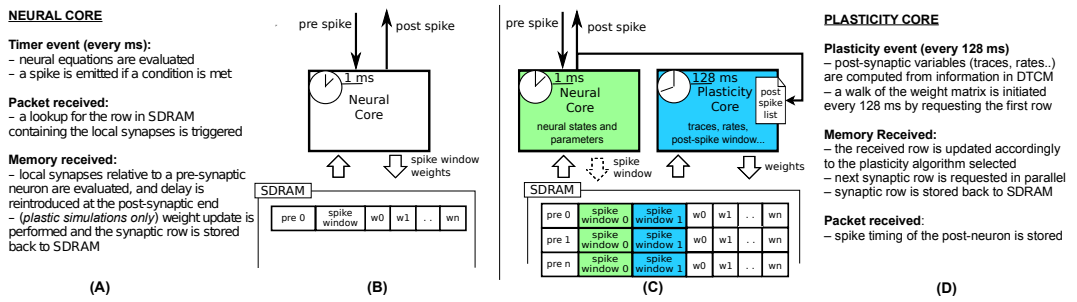


Figure 4.2: (A-B) current STDP implementation on SpiNNaker, following the Deferred Event Driven model (C-D) the proposed novel implementation framework for plasticity implementation.

kernels are parameterizable and can support different classes of neural models and connectivity patterns. Model specification, system mapping and run-time control is obtained through the PArtition and Configuration MANager (PACMAN, [174]), which offers interfaces with two languages extensively used in the neural modeling community: PyNN [175], a simulator-independent specification language, and Nengo [176], the simulation tool implementing the principles of the Neural Engineering Framework.

Figure 4.2(A-B) shows the current implementation of a neural kernel, highlighting the processes involved: every millisecond, a *timer event* triggers the evaluation of the neural dynamics. A spike is then emitted if a configurable threshold of the membrane potential has been reached. Spikes travel as MC packets through routers on the interconnection fabric and are delivered to the destination cores, triggering a *packet event*. Whenever a packet is received, a memory look-up is initiated to retrieve the relevant synaptic information (such as weight, delay, destination neurons on the core, and type of synapse) from SDRAM, where the connectivity matrix, indexed by pre-synaptic neuron, is stored. When the requested data arrives this creates a *memory event*, and the spike is processed by every post-synaptic core. Due to the limited memory available in the ARM cores, the synaptic weights are only locally available to the core right after a memory transfer from DMA has occurred as a consequence of the arrival of a spike. Therefore, the time available for the weight update process is very short; moreover, since delays are reintroduced at the post-synaptic end, the update process relies on information which might concern the future state of the neuron. This has limited the flexibility of previous approaches for implementing plasticity on SpiNNaker.

4.3.1 The Deferred Event Driven Model

The STDP algorithm requires computation whenever a pre spike is received or a post spike is emitted. This causes two relevant issues for the cores running neural simulation on SpiNNaker:

1. Weights are only available in local memory upon the reception of a MC packet signaling that a spike has occurred in one of the pre-synaptic neurons. At the time of a post-synaptic spike such information is stored in SDRAM, which is indexed by pre-synaptic neuron and therefore is not easily accessible for a fast update.

2. A spike packet is delivered to the post-synaptic core as soon as it is emitted, and biological delays (stored in SDRAM as well) are re-introduced by the core modeling the post-synaptic

neuron *after* the relevant information has been retrieved from memory; the delay itself is stored into memory, and can be different for different post-synaptic neurons on the same core. [163]. The weight value is stored in a circular buffer which rotates with the timer event interval, and lumps all the synaptic contributions for one millisecond in a way similar to that described in [149]. The consequence of delaying the input into the future is that when a synapse is processed, the state of the post-synaptic neuron (*e.g.*, its membrane potential or the presence of a post-synaptic spike) is not available.

The *Deferred Event Driven Model* (DED) for computing plasticity was introduced in [177] to circumvent these problems. DED enables computation of STDP at the time when a pre spike is received by deferring the weight update process into the future, until enough information is gathered. Post spikes are collected in a spike window, stored in the local core memory, while pre spikes are stored in SDRAM, along with the rest of the synaptic information. Upon the retrieval of the weights related to a pre spike, these two time windows are compared and weight update is performed. Plasticity is therefore always computed on the *next* pre spike arrival, and only if enough time has passed, to guarantee that all the necessary information is available. This poses restrictions on the pre-to-post firing rates: if a pre-synaptic neuron fires with a low rate, the spike information of the post-synaptic neuron might have already expired. Thus, the algorithm loses a pre-post spike pair, even if they were close in time, if the next pre spike arrives after the expiration of the post spike window. Furthermore, because the algorithm needs to check every spike pair, its efficiency depends on the length of the history and on the number of the pre-post pairs. Such limitations are discussed in [178], where the trade-offs between spike-history, efficiency and correctness are analyzed.

[179] try to address the problem from a different angle, using the *Time To Spike* (TTS) strategy: STDP is computed only upon the reception of a pre spike, using the current membrane potential as a predictor of future spiking activity. By doing so, weight updates can be performed while synaptic information is in local memory, addressing the first of the two problems mentioned above. However, as mentioned earlier, spikes are delivered to the post-synaptic neurons as soon as they are emitted, and the biological delays are reintroduced at the post-synaptic end. This creates errors when using delays, as reported in the original work presenting the TTS approach: the membrane potential used as a predictor of the post neuron firing is the one corresponding to the time of spike *emission* by the pre neuron, rather than that of spike *reception* by the post neuron (after the propagation delay). Such problem makes the TTS algorithm usable and efficient when delays are constant and short, but cannot deal correctly with longer delays. This also creates problems for detecting temporal patterns where delays play an important role [169], such as in the experiments in Section 4.6.2.

4.3.2 The Dedicated Plasticity Core Approach

The previous implementations of plasticity are not limited by the SpiNNaker hardware, but rather by their software implementation. Therefore, we present an alternative approach: instead of having a single core evaluating neural dynamics and plasticity, we divide the job into two parallel processes. One core performs the neural updates and spike integration, while the second core deals with plasticity (see Figure 4.2(C-D)). Plasticity operates as a slower process in the background. It processes the whole synaptic block in SDRAM and the information about spike timing, and modifies the weights according to the chosen plasticity mechanism. The proposed approach

takes inspiration from previous work where plasticity effects are accumulated and evaluated periodically [169, 168, 153, 159]. Plasticity is thus updated less frequently than neural dynamics, which is radically different from the previously described DED model on SpiNNaker.

In our novel approach, the PACMAN mapping tool automatically instantiates a *twin* plasticity core alongside each neural core whenever it detects a neural population with incoming plastic connections. Neural and plasticity cores have access to the same portion of SDRAM through replication, in their local memories, of the look-up tables used to index it. The neural core performs the usual operation that a non-plastic core would perform, thus eliminating all the overheads required by the DED model. The neural core is also in charge of trivially updating a bitmap pre-spike window whenever a pre spike is received, as shown by the dashed arrow in Figure 4.2(C). The plasticity core is concerned solely with the weight update process, which can be performed by walking the local SDRAM weight matrix and computing plasticity at a slower pace. When a neuron in the neural core emits a spike, the corresponding packet is delivered to the plasticity core, and to the post-synaptic neurons as under normal conditions. Because the plasticity and neural core always reside on the same chip, this process does not add overhead to the routing process. This allows to keep track of the post-synaptic spiking history. Here we decided to update the weights every 128 ms and store the spike times with a resolution of 2 ms, as a compromise between performance, platform-specific limitations and precision. Pre-synaptic spikes are stored at the beginning of each synaptic row as spike-history bitmaps. The plasticity process needs to know all the spikes which happened in its considered 128 ms window. This data has been stored by the neural core in one of the spike windows (0 or 1 in the Figure) during the previous 128 ms before the update. For the plasticity core to be able to read this buffer while the neural core is storing the next 128 ms of spikes, we use a double buffer technique: when the plasticity core is reading spike window 0, the neural core is storing the spikes in spike window 1 and viceversa. This has been emphasized in the Figure 4.2(C) by using different color codes for the two different processes. The double buffers contain data for different time slots and therefore do not need to be accessed concurrently by the neural and plasticity core, so there is no need for mutual exclusion or locks. Memory contention is eliminated by the fact that the neural core operates in the *current* 128 ms window, while the plasticity core works in the *previous* 128 ms time window. The same technique used for the spike windows could be used on the whole synaptic matrix to ensure coherency of the whole matrix during the entire simulation. Because this method only switches the pointer used to lookup the data between consecutive plasticity periods, this would not change the approach or performances. Whenever a portion of memory is ready for computation, the request for the next row of the synaptic matrix is issued and weight updates of the current synaptic row are performed, thus masking memory access costs through parallelization. This separation of neural and plasticity operations gives rise to an environment where weight update rules can be easily programmed separately. This leverages the reprogrammability of the general processors used by SpiNNaker and the generality of the event-driven API presented in [173]. While it is worth noting that the difference between neural and the plasticity processes is only in the software running on the ARM cores, they can be thought of as hardware threads. The SpiNNaker software infrastructure does not support threads. If software threads were available, besides the costs related to thread switching, the neural and synaptic update threads would need to split between them the limited local memory (DTCM) and the processor cycles. In SpiNNaker, clock cycles are also limited in order to meet real-time targets. The proposed solution, on the other hand, uses hardware threads (cores), one for neural update and one for synaptic update, with each thread owning all of its local resources. This results in a more efficient use of the available resources. In fact, depending on the relative com-

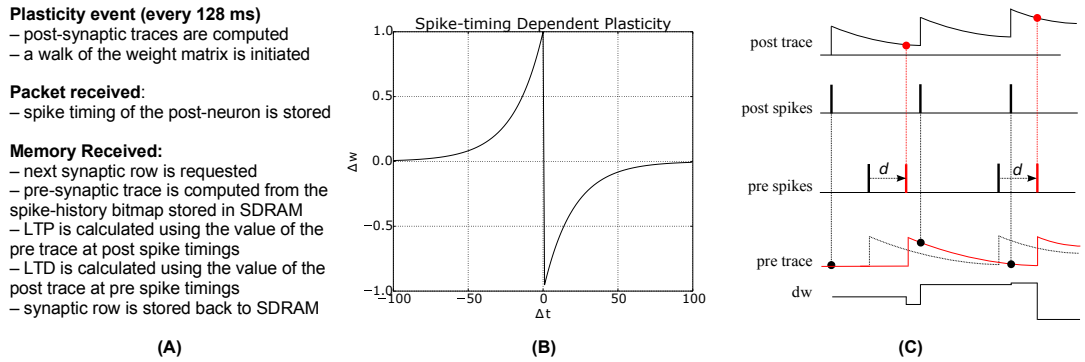


Figure 4.3: (A) Algorithm for STDP learning implementation on the plastic core (B) STDP function (C) Implementation of pair-based STDP with local traces and delays, as suggested by [157]: potentiation occurs at post-synaptic spike times and corresponds to the value of the pre-synaptic trace; conversely, depression happens at pre-synaptic spike times and corresponds to the value of the post-synaptic trace. d represents the delay, reintroduced at the post-synaptic end; black and red lines represent the traces and spike timings when the delay is reintroduced (red) as opposed to using the presynaptic spike time as reference (black).

plexity of the neural and synaptic update processes, the ratio of hardware threads can be adjusted, using N neural update for every M synaptic update threads (cores). The plasticity core has access to the pre- and post-synaptic spike activity history of the previous 128 ms time window; the first is stored in SDRAM and the second one in DTCM. Such information can be used to compute rates, traces, timing differences or other required variables for different learning rules, as shown by the three rules implemented in this work.

4.4 STDP

Derived from biological observations that synaptic plasticity depends on the relative timing of pre- and post-synaptic spikes [125, 126], Spike-Timing Dependent Plasticity (STDP) [129, 131] has become a popular model for learning in spiking neural networks. In its standard form, STDP weight-updates are expressed by the double-exponential form

$$F(\Delta t) = A_+ e^{\frac{\Delta t}{\tau_+}} \qquad \Delta t < 0 \tag{4.1}$$

$$F(\Delta t) = -A_- e^{\frac{-\Delta t}{\tau_-}} \qquad \Delta t \geq 0 \tag{4.2}$$

where $\Delta t = t_{\text{pre}} - t_{\text{post}}$ is the time difference between a pair of pre- and post-synaptic spikes, A_+ and A_- are scaling factors for potentiation and depression, and τ_+ and τ_- are the time constants of the plasticity curves. The weight update rule is illustrated in Figure 4.3. There are different strategies for computing the total amount of weight change after seeing multiple pre- and post-synaptic spikes [157], e.g. by considering only nearest neighbor spike pairs, or summing the weight changes $F(\Delta t)$ for all pairs. Here we adopt a form of STDP proposed by [157] to compute the weight change using local variables in the form of pre- and post-synaptic traces. Each trace x_i has the form

$$\frac{dx_i}{t} = -\frac{x_i}{\tau} + A\sum_{t_i^f}\delta(t - t_i^f) \quad , \tag{4.3}$$

where x_i is the value of the trace for neuron i, A is the amplitude by which the trace increases with each new spike at time t_i^f, and τ is the exponential decay time constant. The concept is illustrated in Figure 4.3: potentiation occurs at post-synaptic spikes, using the value of the pre-synaptic trace as the weight increase; conversely, depression happens at pre-synaptic spike times, and reduces the weight by the value of the post-synaptic trace.

4.4.1 Methods: Implementation of STDP on the Plasticity Core

The plasticity core is in charge of computing all traces, using the spike timing information collected during the simulation. Weight changes are then computed by walking through all the synaptic block. The pre-trace is computed every time a portion of memory is received through a DMA process using the information in the spike window, while the post trace is computed at the beginning of each plastic phase starting from the spike history bitmap collected during the packet received callback. Traces can have longer time scale than the plasticity window, as the exponential filtering is updated at the beginning of each phase, and the previous value of the exponential filter carries over from one plasticity window to the next. Delay needs to be reintroduced at the post-synaptic end, and can be used to compute the amount of shift required to correctly compute weight de/potentiation, as shown in Figure 4.3, where the black part shows the spike timing and traces using the presynaptic spike time as the reference, while the red part shows how this reference is shifted once delay has been reintroduced. Not considering the delay generates substantial errors in the weight update.

A simple experiment in which STDP, implemented with the above scheme achieves synaptic potentiation and depression is shown in Figure 4.4 (A) and (B). The final part of the Figure presents a classical experiment where a plastic neuron can reduce the latency of its firing to a repeatedly presented pattern [131, 180]: a (red) neuron receives connections from 10 inputs neurons (blue) which fire at 2 ms from each other; during the first repetition all the 10 input neurons are required to make the target neuron fire. After repeated presentations, due to potentiation, only three input neuron spikes are needed to elicit activity in the post neuron, which responds with a lower latency to the onset of the pattern.

4.4.2 Results: pre-post pairing using a teacher signal

In Figure 4.5 we reproduce results of a classical stimulation protocol for potentiation induced by pre-post synaptic pairing. The network comprises a *stimulus* population and a *target* population, each separately driven by two different Poisson sources emitting spike bursts at high frequency (350 Hz) for short periods of time (20 ms). Both populations also receive independent background noise. The Poisson and noise source populations are interconnected with a one-to-one connectivity pattern to their respectiv inputs and outputs. The stimulus and the target populations are interconnected with a 50% probability.

At the beginning of the simulation, external stimulation coming from the stimulus population is not strong enough to trigger activity in the target post-synaptic population ($0 \leq t \leq 1500$ ms). Afterward ($1500 \leq t \leq 3000$ ms) the stimulus and target populations are stimulated together by

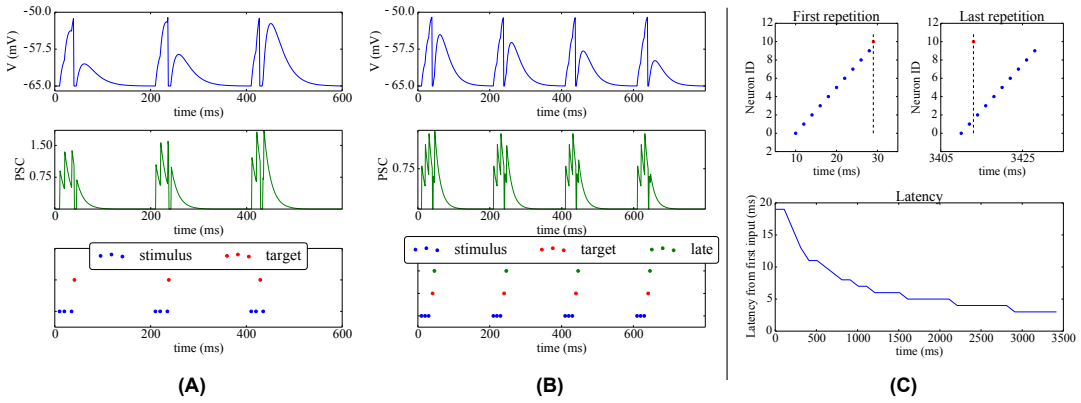


Figure 4.4: Shift of post-synaptic firing onset via STDP. (A) Potentiation: The spike raster plot (bottom) shows that at the beginning of the stimulation 3 input spikes (blue) are needed to make the target neuron (red) fire; after 400ms, potentiation has made the synapse strong enough so that the post-synaptic neuron fires after only 2 spikes. This is also visible in the membrane potential (top) and post-synaptic currents (PSC; middle). (B) Depression: (bottom) The green neuron is made to spike consistently after the target (red) neuron, hence its weight gets depressed, as can be observed by its decreasing contribution in the membrane potential (top) and PSC (middle). (C) Reduced spike latency: at the beginning of the simulation (upper-left panel) 10 spikes from 10 different input neurons (blue) are needed to make the post-synaptic (red) neuron fire; after repeated stimulation (upper-right-panel), potentiation via STDP makes the red neuron fire already after 3 spikes, hence firing closer to the pattern's start, which is also shown by the latency plot (bottom panel).

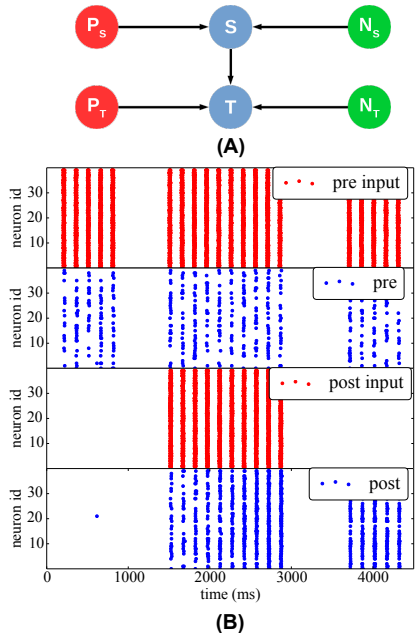


Figure 4.5: STDP with a teacher signal: (A) Network structure: two different Poisson spike sources (red) are used as supervisor signals to individually stimulate the *Stimulus* (top) and *Target* (bottom) populations at different times (blue), which also received noise from two separate sources (green). (B) Initially (between 0 and 1000 ms), only the pre-synaptic population is stimulated, but the synaptic weights are weak, thus the resulting spikes (blue) do not elicit post-synaptic spikes. Between 1500 and 3000 ms, both populations are stimulated with a 10 ms time difference, such as to induce synaptic potentiation. The effect can be seen between 3500 and 4500 ms, when the teacher signal for the post-synaptic population is removed: after the potentiation the pre-synaptic spikes are able to drive the post-synaptic neurons by themselves.

their respective Poisson inputs, so that the target population spikes 10 ms after the stimulus population, hence inducing potentiation. Finally, for $3500 \leq t \leq 4000$ ms, the Poisson process feeding the post-synaptic population is removed, and the post-synaptic population is only stimulated by inputs from the pre-synaptic population. It can be seen that because of the induced potentiation, the pre-synaptic input is now strong enough to make the target population fire without any supervisor input.

4.4.3 Results: balanced excitation

[131] have shown that STDP can establish a state of balanced excitation in the post-synaptic neuron, which makes it more likely to fire with a controled output rate in response to fluctuations in its input. This is achieved by competition between the synapses that project onto the post-synaptic neuron, induced by STDP. The characteristic effect described by [131] is that STDP creates a bimodal distribution of input weights, pushing them either towards the minimum or maximum values, and creating groups of *strong* and *weak* synapses. In Figure 4.6 we simulate a group of 1000 input neurons, firing independently according to a Poisson process at 20 Hz, and projecting onto a single output neuron. The weights are initialized uniformly, and then undergo STDP.

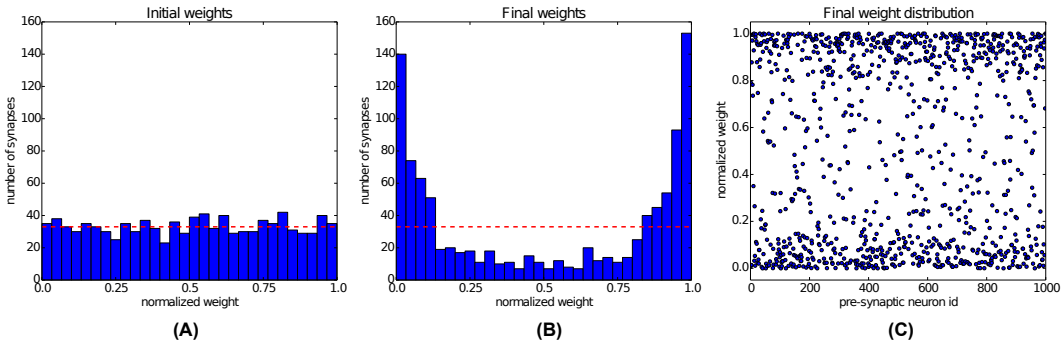


Figure 4.6: Competition between synapses undergoing STDP: In the experiment introduced by [131], 1000 pre-synaptic, uncorrelated pre-synaptic neurons, firing at a Poisson-rate of 20 Hz, project onto a single post-synaptic neuron. (A) Initial uniform weight distribution before plasticity. (B) After 300 seconds of stimulation STDP has divided the synaptic weights into *weak* and *strong* ones, thereby regulating the activity of the post-synaptic neuron. The red line shows the mean of the initial weights. (C) Scatter plot of the final weight distribution.

After 300 s of simulation, the distribution of synaptic weights in Figure 4.6 shows clearly the characteristic separation into two groups of very different strengths.

4.5 BCM

The *BCM rule*, named after their inventors Bienenstock, Cooper, and Munro [124], is a rate-based synaptic plasticity rule, introduced to model binocular interactions and the development of orientation selectivity in neurons of the primary visual cortex. The BCM rule is based on Hebbian principles, but introduces synaptic competition by correlating the pre-synaptic rate with a non-linear function of the post-synaptic rate. In its simplest form the BCM rule computes this non-linearity as the product of the post-rate with its deviation from the mean post-synaptic activity (see Figure 4.7(B)):

$$\frac{dw}{dt} = [r_{post}(r_{post} - \theta)r_{pre}]\delta - \varepsilon \cdot w \quad . \tag{4.4}$$

Here w denotes the synaptic weight and dw/dt its change, r_{post} and r_{pre} are the firing rates of the pre- and post-synaptic neurons, θ is the modification threshold, which is computed here as the mean firing rate of the post-synaptic neuron, δ is a learning rate, and ε a weight-decay parameter. If r_{post} exceeds the mean firing rate θ, the weight is potentiated; conversely, for lower activity ($r_{post} < \theta$) the weight is depressed. The learning rate parameter δ can be used to normalize the magnitude of the synaptic weight change according to the neural model used. Many variations of the BCM rule have been studied since its introduction, using different kinds of non-linearities, but here we study only the basic version from [124].

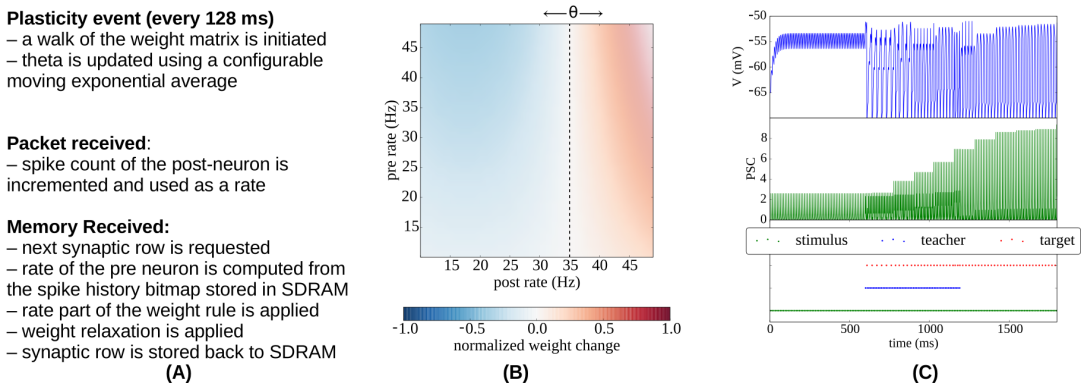


Figure 4.7: Implementation of BCM plasticity. (A) Algorithm for BCM learning on the plasticity core. (B) Illustration of the BCM rule: normalized weight change as a function of the pre- and post-rates, with $\theta = 35$ Hz. (C) Potentiation experiment using a teaching signal: when stimulation is paired with a teacher signal that forces the post-synaptic neurons in the target population to fire, the weights get potentiated and become strong enough to drive the post-synaptic neuron.

4.5.1 Methods: Implementation of BCM on the Plasticity Core

Since the BCM rule only requires firing rates, the plasticity core just has to increment a counter whenever a post-spike is received, and to use a low-pass filtered version of the rate. Analogously, when processing a row relative to an afferent neuron, the number of spikes received during the previous phase is used to update the pre-synaptic rate information. At the end of each plasticity phase θ (the threshold parameter representing the mean rate) is updated using a configurable exponential moving average, and the pre spike windows are reinitialized.

In Figure 4.7(C) we show a classical potentiation protocol using the BCM rule. For the first 600 ms the target population is only receiving spikes from the stimulus population, but the weights are too weak to cause firing in the target population. Between 600 and 1200 ms, a teacher population is activated which is strong enough to drive the target population, thereby potentiating also the simultaneously active stimulus-target connections. Afterwards, when the teacher population is switched off, the stimulus population alone is able to drive the target population without teacher input.

4.5.2 Results: Emergence of orientation selectivity with BCM

The BCM rule has been originally proposed in [124] to explain how neurons in the primary visual cortex can acquire their feature selectivity from sensory stimulation. As a test of our implementation of BCM on SpiNNaker we replicate a simple neural network with lateral inhibition which undergoes plasticity while receiving monocular visual input in the form of oriented bars.

The network consists of 2 layers, an input layer which comprises 16×16 neurons and an output layer with 4 neurons. Each neuron in the input layer projects, in an all-to-all fashion, to the output neurons. All synapses are initialized with random weights and delays. Each neuron in the output layer has an inhibitory projection to every other neuron, forming a network of lateral antagonism [181]. The aforementioned connectivity pattern matches anatomical data, for example

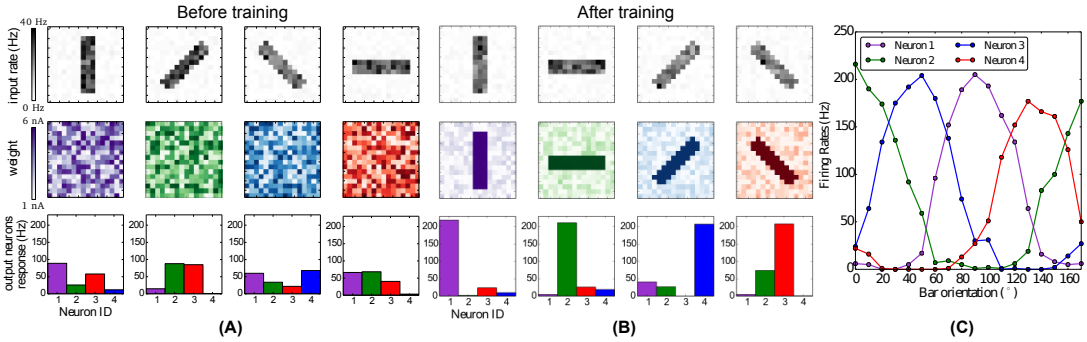


Figure 4.8: Emergence of orientation selectivity with the BCM learning rule. (A) and (B): the top row represents the input stimuli bars presented in different orientations, the total firing rate for each stimulus is 1,000 Hz; the middle row shows the weight matrix for the output neurons, with (A) random initial weights and (B) results after the training, where it can be observed that neurons have developed their receptive fields according to the input stimulation; the bottom row shows the firing rates of the output neurons, where each color codes for a different neuron which has learned a preferred orientation. (C) Orientation tuning curves obtained by rotating a horizontal bar counter-clockwise with a step size of 10 degrees.

the *lateral plexus* of the Limulus's eye, as originally found by [182]. [124] themselves point out that no selectivity is achieved without lateral inhibition.

For this experiment four images of oriented bars are used as input stimuli, each rotated by 45 degrees. Bars are 3 pixels thick and 12 pixels in length, and the intensity of each pixel is a random value between 0.8 and 1.0. Each pixel is converted to Poisson spike trains, in order to simulate spikes coming from the retina or LGN. The firing rates are proportional to the value of the pixels, while all firing rates are scaled such that the input layer generates approximately 1,000 spikes per second. During the simulation each orientated bar is presented to the network in a random order for 1 second and for 80 repetitions. Learning takes place in the synapses between the input and output layer, while the inhibitory synapses in the output layer are static and set to a weight of -9 nA.

The results are summarized in Figure 4.8. Figure 4.8(A) shows that the weights and neuronal responses to input stimuli are initially random. At the end of the simulation, Figure 4.8(B) shows that each output neuron has developed via BCM plasticity a receptive field that corresponds to one particular orientation. In Figure 4.8(C) we show the orientation tuning curves of each neuron, measured by rotating the stimulus bar counter-clockwise in 10 degrees steps. The results show that each neuron has successfully learned to respond best to one preferred orientation, which is in line with previous modeling studies and experimental and anatomical data [183, 184].

4.6 Voltage-gated STDP

[142] have presented an STDP rule that is triggered by the arrival of pre-synaptic spikes, and in which the change in synaptic efficacy is a function of post-synaptic depolarization and of an internal variable at the spike arrival time. The rule is motivated by the necessity to design learning

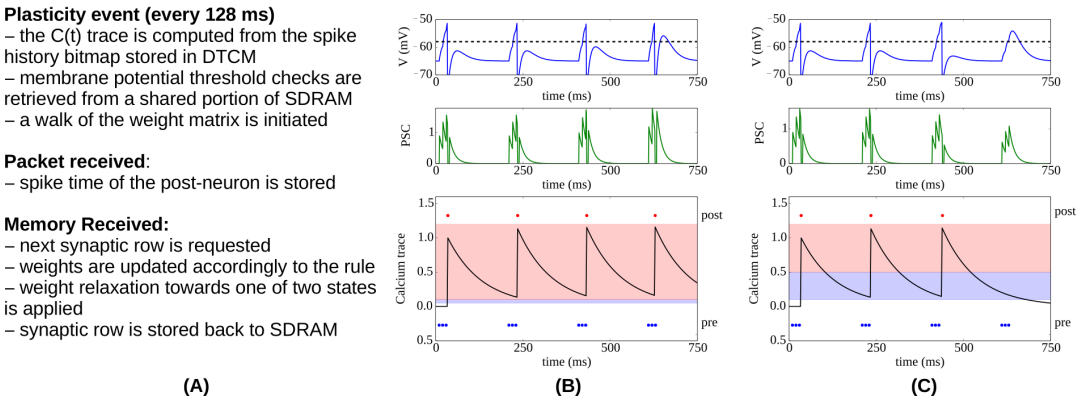


Figure 4.9: Implementation of voltage-gated STDP. (A) Concept for implementing voltage-gated STDP on the plasticity core. (B) Example of synaptic potentiation: three pre-synaptic spikes (blue) arrive while the membrane potential is greater than θ_V and $\theta^h_{down} \leq C(t_{pre}) < \theta^h_{up}$ (red-shaded area in bottom row). Initially, the post-synaptic neuron (red) fires after the third spike, after potentiation only two spikes are needed to make the target neuron fire. (C) Depression example: three pre-synaptic spikes (blue) arrive while the membrane potential is less than θ_V and $\theta^l_{down} \leq C(t_{pre}) < \theta^l_{up}$ (blue-shaded area in bottom row). After depression takes place, the post-synaptic neuron (red) no longer fires after receiving the 3 input spikes.

rules which are at the same time biologically plausible, but also compatible with implementation constraints on neuromorphic devices. Several studies have demonstrated the ability of the learning rule to discriminate complex spatio-temporal patterns [185, 186, 187], even if the synapses are allowed to take on only one of two stable states. Every time the post-synaptic neuron emits a spike an internal variable $C(t)$, representing calcium concentration due to back-propagating action potentials, is incremented by a value J_C and then decays with a time constant τ_C according to the dynamics described by

$$\frac{dC(t)}{dt} = -\frac{C(t)}{\tau_C} + J_C \sum_i \delta(t - t_i) \quad , \tag{4.5}$$

where t_i are the post-synaptic spike times. Potentiation and depression happen only if $C(t)$ is in an appropriate interval $[\theta^h_{down}, \theta^h_{up}]$ for potentiation and $[\theta^l_{down}, \theta^l_{up}]$ for depression. Post-synaptic membrane depolarization $V(t)$ influences this plasticity rule, triggering potentiation (or depression) only if the membrane potential of the post-synaptic neurons is higher (lower) than a threshold value θ_V at the time of arrival of a pre-synaptic spike (t_{pre}). Modification of the synaptic efficacy w can then be summarized by the following equations:

$$w = w + a \quad \text{if} \quad V(t_{pre}) > \theta_V \quad \text{and} \quad \theta^h_{down} \leq C(t_{pre}) < \theta^h_{up} \tag{4.6}$$

$$w = w - b \quad \text{if} \quad V(t_{pre}) \leq \theta_V \quad \text{and} \quad \theta^l_{down} \leq C(t_{pre}) < \theta^l_{up} \tag{4.7}$$

where a and b represent the constant weight increase and decrease values respectively.

If none of the conditions in (4.6) and (4.7) are met, or if no spike is received in the period of time considered, then the weight drifts towards one of two stable values (w_{min} and w_{max}). The

direction of the drift is determined by comparing the current weight w to a threshold θ_W, and speed of the drift towards the minimum and maximum stable states is determined by the constants α and β respectively. This leads to the following dynamics:

$$\frac{dw(t)}{dt} = \alpha \qquad \text{if} \quad w(t) > \theta_W \tag{4.8}$$

$$\frac{dw(t)}{dt} = -\beta \qquad \text{if} \quad w(t) \leq \theta_W \tag{4.9}$$

4.6.1 Methods: Implementation of voltage-gated STDP on the plasticity core

The voltage-gated STDP rule needs further information from the post-synaptic neuron, as the membrane potential gates potentiation or depression. The cores communicate this information as means of shared memory in SDRAM, using a double buffer technique so that they always work on different phases. This induces a slight overhead in the neural core, which has to perform the check against θ_V and saves the result for each millisecond in a bitmap stored in memory. The plasticity core retrieves the results of the comparison at the beginning of each plasticity phase, and uses them in the weight update process. At the same time the function $C(t)$ is computed starting from the post neuron spike timings, similarly to computing the STDP traces.

The basic dynamics of this voltage-gated STDP rule are shown in Figure 4.9: The bottom row shows the trace of the calcium variable $V(t)$, which is increased by J_C whenever the post-synaptic neuron fires, and then exponentially decays with time constant τ_m. The central part shows the potentiation of a synapse, because here the pre-spikes arrive when $V(t_{pre}) > \theta_V$ and $\theta^h_{down} \leq C(t_{pre}) < \theta^h_{up}$), and thus fewer spikes are needed to drive the target neuron. Conversely, on the right we observe the depression of a synapse, because pre-spikes arrive when $V(t_{pre}) \leq \theta_V$ and $\theta^l_{down} \leq C(t_{pre}) < \theta^l_{up}$). After depression, the synaptic input is too weak to make the target neuron fire.

4.6.2 Results: Learning temporal patterns

To verify our implementation of the voltage-gated STDP rule by [142], we implemented the model by [188] for learning temporal structures in auditory data, which has originally been implemented on a neuromorphic chip in [144]. The study focused on learning dynamical patterns in the context of a sound perception model by tuning auditory features through presentation of stimuli and learning using the STDP rule implemented in VLSI.

The proposed network learns to respond to particular input timing patterns. The network comprises 3 layers of tonotopically organized frequency channels, representing different positions on the basilar membrane. The first layer A represents a spiking signal produced by an artificial cochlea (such as the one in [189]); each neuron in the A layer projects to a neuron in 2 layers, B_1 and B_2 through excitatory synapses, while B_1 projects to B_2 through inhibitory synapses. Each neuron in B_2 also receives plastic connections from all the neighboring B_1 neurons, with delays proportional to the distance, as shown in Figure 4.10. Since delays are programmable in SpiNNaker we incorporated them directly in the B_1 to B_2 connection, and not through a separate neural population as in the original model. This delay property is essential for learning: correlation between the delayed feedback arriving from other B_1 neurons to the B_2 neurons is detected by the plasticity rules, and it controls synaptic potentiation and depression by coincidence detection. To implement the model on SpiNNaker while coping with the 1 ms time resolution used in the current neural kernels we

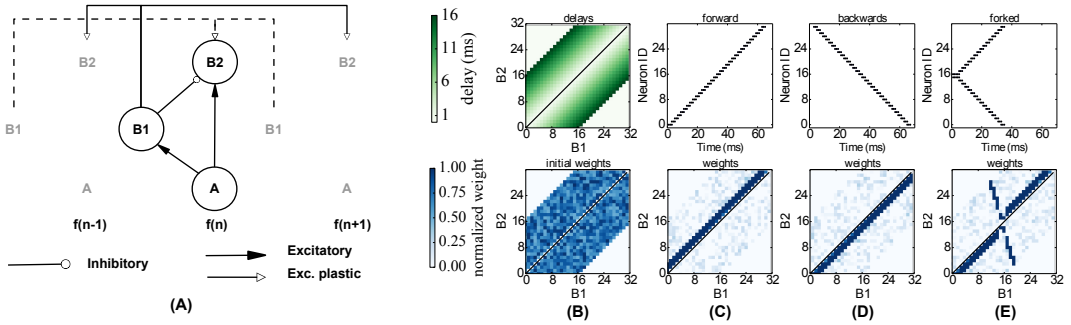


Figure 4.10: Learning temporal patterns with the model by [188] and [144]. (A) Network structure: each frequency channel f comprises three neurons A, B_1 and B_2, and is connected to other frequency channels (dashed in figure), arranged in a tonotopical way, with distance-based delays and plastic connections. (B) Delay matrix (top) and example of initial random weight matrix (bottom). (C-E) Input spikes (top) and resulting weight matrix (bottom) after learning for a forward, backwards and forked frequency swipe respectively.

multiplied all the time quantities by 10. For learning we use the same three input patterns that were used in the original model (see Figure 4.10): Pattern (C) is a forward frequency sweep, where every frequency (and therefore every A neuron) is activated in order, with a short delay between one presentation and the next. For pattern (D) we perform the same frequency sweep, but we move backwards through the frequency space. Finally for pattern (E) we perform a forked frequency sweep, starting from the middle frequency. We present the stimulus multiple times to the network and analyze what it has learned by examining the B_1/B_2 weight matrix. The results are presented in Figure 4.10, and can be compared with the results in Figure 7 and 8 in [144]. After repeated presentations of the target patterns, the weight matrix, initialized randomly, converges to a state where it is only sensitive to the spike-timing pattern presented, by coincidence detection through delay lines.

4.7 Performance analysis and discussion

In [190], the authors describe an STDP variation of the DED which follows the strategy proposed in [157] by storing traces in SDRAM, rather than performing spike pairing as proposed in [178]. The authors evaluate the performance of their implementation as well as the one present in the stable release of the SpiNNaker software package[1] in terms of synaptic events processed per second, as done in [191] and [192]. They do so by feeding a leaky integrate-and-fire population of 50 neurons with a neural population of variable size that produces spikes at ≈250 Hz, according to a Poisson process, with a 20% connection probability. They report that their implementation of plasticity is capable of handling around 500K synaptic events per second per core (using 150 input neurons), while the original SpiNNaker implementation is limited to 50K events.

We adopt a similar strategy to evaluate our plasticity algorithms, but in more stringent conditions, and with a larger connectivity range. Rather than testing a single core we test a full chip (16 cores). In this way, we can also evaluate the effects of memory contention between different

[1] `https://spinnaker.cs.man.ac.uk/tiki-index.php?page=SpiNNaker+Package+(quarantotto)`

cores, as memory access can be a bottleneck for simulations on SpiNNaker. We model a population of 800 neurons in a single SpiNNaker chip (8 cores modeling neurons and 8 cores dedicated to plasticity) fed by an input Poisson neural population of 150 neurons with a variable rate, and and measure the maximum firing rate at which the simulation can run in real time. We take as a starting point the connectivity levels reported by [190] (20% interconnection probability, $150 \times 50 \times 0.2$ synapses, for a total of 1,500 per core and 24,000 per chip if considering 16 cores) and increase the connectivity level up to 100% (7,500 synapses per core, 120,000 per chip). This results in synaptic rows which are 5 times longer, as every pre-synpatic neuron is connected to every post-synaptic neuron in each core, rather than only 20% as in the original experiment. We then scale the model further up by adding more pre-synaptic neurons so as to reach a total of 156,000 synapses. The performance analysis of the algorithms proposed in this work uses the same leaky integrate-and-fire current based neuron. To be able to scale the rate while maintaining the post-synaptic activity constant, we set all the weights and all the weight increments in the plasticity rules to 0, similarly to the approach in [192]. This means that plasticity is normally computed, but the weight is clipped to 0 and stored back in SDRAM. Such values are set at runtime and cannot therefore by optimized by the compiler; we have also ensured that setting these values to 0 would not bypass part of the code by removing some optimization tests (like not updating weights which do not change), thus ensuring that the code behaves in our test case as the worst possible real case. Post-synaptic activity is induced by feeding the leaky integrate-and-fire neurons with a current inducing an activity of ≈ 22 Hz.

We check if at any moment any core is lagging behind real time as this would make the simulation incorrect and unrepeatable. We also check if a walk through of the weight matrix is completed before the end of the plasticity period or, in other words, if the plasticity process is finished before the next one starts, as overlapping in this sense is not possible when operating in real time. This allows us to measure the maximum number of synaptic events that can be handled in real time by a single SpiNNaker chip, using the three learning rules proposed in this chapter (STDP, BCM and voltage-gated STDP), and to understand if the performance is limited by the neural or the plasticity core.

Results are shown in Figure 4.11; for each given connectivity level (number of synapses) pre-synaptic firing rates are increased until the limit after which real-time simulation is no more possible. Each point of the plot hence represents the limits of the approach for a given connectivity, for each of the plasticity rules implemented. From the Figure it can be observed that the three learning rules implemented within this framework have similar performances untill the limit of 96,000 synapses (corresponding to scaling up to 80% connectivity the model by [190]). This is due to the fact that, up to that point, all three learning rules are limited by the neural cores lagging behind real time, rather than by the plasticity process taking too long. Such limit peaks just below 1,5 million synaptic events per second per core for all three rules (23 million events for the full chip). In a non-plastic performance analysis, [192] measured a maximum throughput of ≈ 2.38 million synaptic events per second per core. After this connectivity level the complexity of the two STDP models (standard and voltage-gated) becomes the limiting factor, and a complete walk of the synaptic matrix is not possible anymore within the 128 ms period used in this work. The BCM algorithm is not affected by this, as the algorithm is computationally less intense, and keeps improving above 1,6 M synaptic events per second per core. The decay in performances reflects the complexity of the algorithm considered: standard STDP, being more complex than the voltage-gated version, has a sharper decrease in performances.

When comparing these scenario results with the previous plasticity models based on the DED

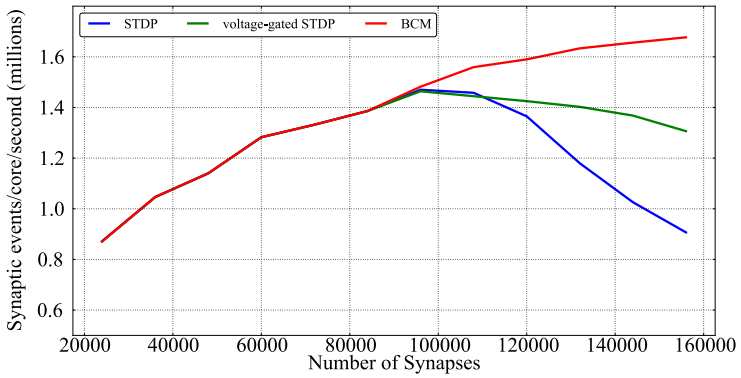


Figure 4.11: Performance evaluation of the three learning rules in terms of synaptic event processed per core per second as a function of different number of synapses.

by [178] and [190] (around 50k and 500k synaptic events per second per core respectively in the 20% case - the leftmost part of Figure 4.11), it must be remembered that these algorithms work with a 1 ms spike-window resolution, while the experiments proposed in this work have adopted a resolution of 2 ms. Also the former algorithms might lose spikes, while in the approach presented here the contributions from *all* the spikes are accumulated (or, in other words, no spike is lost).

While our approach was designed for maximal flexibility, there might be tradeoffs in terms of efficiency for some scenarios, depending on connectivity and firing rates. One limitation of our approach is, for example, that every plasticity event triggers an update of the complete synaptic matrix. For the rules proposed in this chapter is not possible to selectively update only some rows. For pre/post sensitive rules (such as STDP) destinations are encoded in the synaptic row, which is stored in SDRAM, so it is not possible to know if a pre neurons connect to a post neuron which has fired (thus inducing LTP) before retrieving the row itself. In rate based models such as BCM, where the firing rate is considered as a moving average, the absence of spikes is not sufficient to ensure there is no plasticity in act. Finally for rules with relaxation towards one (BCM) or multiple (voltage-gated STDP) states require a weight update even in the absence of a spike.

Since in our implementation plasticity updates occur every 128 ms, pre-synaptic firing rates should be at least on the order of $\approx 7-8$ Hz to avoid having to update silent synapses regularly. In scenarios with lower firing rates, a purely event-driven update would be more efficient. However, a main motivation for our approach is to ensure real-time performance, even in situations with momentarily high load, e.g. if multiple neurons are firing in bursts. Such scenarios are common when using natural inputs with coincident input spikes or models with oscillatory background signals. In such cases the plasticity core approach offers greater flexibility to process plasticity in real time: instead of having to process neural and synaptic updates of all simultaneous spikes within the 1ms time step of the neural core, which might be challenging for complex plasticity rules or for complex neural models, our approach accumulates events over the longer time window of the plasticity core.

This decoupling enables the neural cores to maintain the real-time constraints, and opens up new possibilities for trade-offs to reduce the load on the plasticity cores if necessary. The simplest possibility is, as in the DED model, to lower the number of neurons simulated by each neural core (and therefore also by its associated plasticity core). Other options, although not implemented in

the first proof-of-concept presented in this work, are possible. For the initial results presented in this work we maintain a 1:1 ratio between neural and plasticity cores, but this will likely not be optimal for all scenarios. When looking at Figure 4.11 it can be see that the two STDP models show a sweet spot for performance at around 80% connectivity. Before such maximum the performance is limited by the neural cores, while after that is the plasticity core which is not able to keep up with the real-time requirements. An interesting alternative would be to allocate more plasticity cores to a single neural core, and adapt the *plasticity:neural* core ratio according to the network characteristics and to the computational complexity of the neural and plasticity algorithms and the associated workload.

A limiting hardware factor for any implementation of plasticity on SpiNNaker is the memory bandwidth, because rows of the synaptic matrix need to be written back to SDRAM. It was shown in Fig. 9 of [172] that writing is the main bottleneck, since the read bandwidth is twice as high. Our approach reduces the write load, since rows are only written back to SDRAM at most once every plasticity interval, rather than once every pre-synaptic spike as in the DED model. This means that, for example, if pre-synaptic neurons are firing at 24 Hz each synaptic row would be transferred back to SDRAM 24 times per second using the DED model, but only 8 times with our approach.

Finally another possibility is to increase the duration of plasticity intervals, which increases the time available for computing the updates, but comes at the cost of larger memory requirements for storing traces in the core-local DTCM. For long plasticity intervals this might grow beyond what can be stored in DTCM (64 Kbytes for each ARM core, of which some space needs to be reserved for other parameters and buffers). The capacity can be increased by lowering the precision for storing the traces, or using a coarser time resolution. All these possible trade-offs, although not fully explored in this initial work, show the versatility of the approach, which can be adapted to different situations and modeling needs, and constitutes one of its key features, as discussed in the last Section.

4.8 Discussion

Current research on understanding the relationship between the local electrochemical processes of synaptic plasticity and their manifestations as high-level behavioral learning and memory is increasingly relying on theoretical modeling and computer simulations [193]. Because of the great diversity of plasticity phenomena observed in biology and the resulting diversity of proposed mathematical models, as well as the computational complexity of spiking neural network simulations dominated by the costs of synaptic processing, it is necessary to create simulations tools that provide both the flexibility to try out new models easily, and the speedup of specialized hardware. This meets the demand of increasingly large neural network simulations, both for studying brain function, and for applications in artificial intelligence [194]. SpiNNaker has proven to be a well-suited platform for massively parallel large-scale simulations of spiking neural networks, and is flexible enough to let researchers implement and test their own computational models in standard programming languages. The previous Deferred Event Driven Model of handling events in SpiNNaker has made it difficult to implement plasticity rules with arbitrary triggering events (pre-, or post-synaptic, or at regular time intervals), rules which depend on third factors available only at the post-synaptic neuron, or plasticity in networks with variable axonal delays. We have presented here a framework which uses the modular architecture of SpiNNaker and delegates weight updates

to dedicated plasticity cores, while the network simulation operates on the remaining neural cores. We have shown that a variety of commonly used plasticity rules can be exactly replicated on this framework, with a greatly increased capacity of processing plasticity events in real-time, by running experiments on a 4-chip SpiNNaker board. The separation of neural from plastic concerns is the feature that enables the great flexibility of the architecture. The two cores work in parallel on different time scales and phases, and the plasticity core has all the information to compute plasticity for the recent past, can access the weight matrix shared with the neural core, and any other information that can be passed through means of shared memory, e.g. membrane potentials and spike timings of the pre- and post-synaptic neurons. All this information can be pre-processed before plasticity is computed, which allows e.g. the computation of rates in an otherwise spike-based simulation. The architecture can be configured easily, using PyNN scripts. This standard, high-level neural language makes it easy to integrate and explore new learning rules into the SpiNNaker architecture.

The approach presented in this chapter is tailored to SpiNNaker and to its specific architecture, design and constraints. Nonetheless the same principles could be applied to other digital-analog hybrid architectures, where efficient neural simulation could be realized on one neuromorphic chip, whereas complex plasticity rules could be realized off-chip on computers or FPGAs. Regarding GPUs it appears to be more favorable to let each kernel perform the same operation following the SIMD paradigm. [195] sequentially use two different kernels, one for neural updates and one for applying plasticity updates. Such kernels do not run in parallel on the same GPU, but serially. This does not constitute a problem when running accelerated simulations, which is the common case for GPUs, but can raise difficulties when running in closed-loop real-time scenarios, as in neurally inspired robotics [196]. In fact concurrent kernel execution is a feature that has only recently been introduced in GPUs, with the NVIDIA Fermi architecture. Using such technique, a plasticity and a neural kernel could be instantiated concurrently on the same GPU, in a similar way to what is done in our approach. Memory access patterns, and the possibility of accessing contiguous portions of memory is a key factor when programming a GPU [197]. It could be speculated that applying an approach like the one proposed in this work would have the benefit of guaranteeing memory coalescence, as the synaptic matrix is sequentially accessed when walking through it. Multi-core or cluster architectures could also in theory benefit of separating neural simulation and plasticity, running either on different threads or on different cores, and with different time scales. However, clusters are equipped with more powerful processing units than SpiNNaker, so computing neural and synaptic updates in different cores could introduce unnecessary overheads and synchronization difficulties, particularly regarding memory bandwidth and access patterns.

In our experiments we have deliberately chosen to reproduce classical results, in order to compare the run-time performance of the novel framework to previous implementations of plasticity on SpiNNaker. The examples of BCM, STDP, and voltage-gated STDP learning provide templates for constructing further experiments with rate-based, spike-timing-based, and voltage-dependent learning rules. Our approach can be easily extended to include additional third factors to modulate plasticity, e.g. neuromodulators [133, 136], or weight-dependency [145, 157], can model homeostatic effects [198], or handle different synaptic delays [199, 89]. It can also combine different models of plasticity in one simulation, a feature which is used in several recent models, where network function arises from the interaction of different synaptic plasticity rules that are specific to particular cell types [200, 147, 201, 202]. In fact, we have provided a tool that should be general enough to model long-term potentiation rules, but is not restricted only to phenomenological ones. Other biological structures *i.e.* glial cells are considered to have a fundamental role in plasticity,

and can enhance learning capabilities [203]. The plasticity core, by leveraging this functional segregation already present in biology, is a natural candidate to model such structures.

The results presented in this work and the possibilities opened by this approach point to the efficiency and to the generality of the framework introduced: a modular, flexible and scalable tool for the fast and easy exploration of learning models of very different kinds on the parallel SpiNNaker system.

Chapter 5

Breaking The Millisecond Barrier On SpiNNaker: Implementing Asynchronous Event-Based Plastic Models With Microsecond Resolution

The spikes produced by neuromorphic sensors usually have a time resolution in the order of microseconds. This high temporal resolution is a crucial factor in learning tasks. It is also widely used in the field of biological neural networks. Sound localization for instance relies on detecting time lags between the two ears which, in the barn owl, reaches a temporal resolution of 5 *microseconds. Current available neuromorphic computation platforms such as SpiNNaker often limit their users to a time resolution in the order of milliseconds that is not compatible with the asynchronous outputs of neuromorphic sensors. To overcome these limitations and allow for the exploration of new types of neuromorphic computing architectures, we introduce a novel software framework on the SpiNNaker platform. This framework allows for simulations of spiking networks and plasticity mechanisms using a completely asynchronous and event-based scheme running with a microsecond time resolution.*

Contents

5.1 Introduction

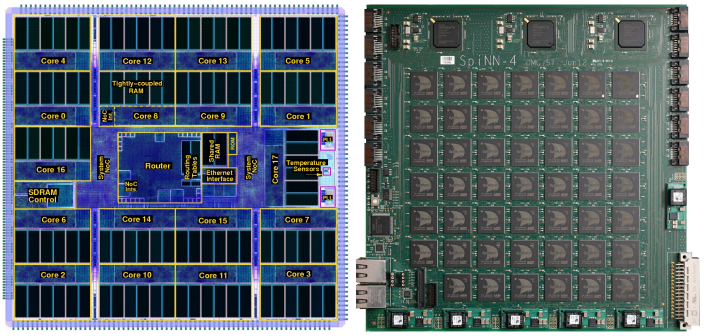
The ability of neurons to fire stereotypical action potentials with a very high temporal resolution, observed both *in vivo* [204, 205] or *in vitro* [206], points at the importance of temporal precision in neural coding. [207] analyzed the temporal properties of neurons in the medial temporal (MT) area of monkeys by using single-cell recordings. They demonstrated that 80% of the cells in area MT are capable of responding with a jitter of less than 10 ms, while the most precise cells have a jitter of less than 2 ms. At the sensory end, ganglion cells are known to emit action potentials with a time resolution in the range of milliseconds [208, 209]. However some neurons can encode signals which are even faster than their own dynamics. [210] suggests that dendritic trees can be responsible for high-temporal coincidence detection with time constants faster than the ones of their neural membrane. [129] have addressed this apparent paradox by using as an example the auditory system of the barn owl. They can locate a target sound with a precision of a couple of degrees, corresponding to a resolution of 5 μs. They show how neurons, which have synaptic and membrane time constants orders of magnitude larger, can phase-lock and respond to signals arriving coherently in a short time window. Learning plays a crucial role in shaping such connectivity, and in tuning cells to preferential input phases. Plasticity also mediates cross-modal interactions to give rise to the precise sound localization system in the owl [211, 212], where visual and auditory inputs are combined together to shape the neural circuitry responsible of such great temporal precision.

As the extent of the precision needed by models of spiking neural networks is still a matter of debate, having neural platforms capable of rapidly acquiring and generating sensory data at high temporal resolution becomes a valuable asset for scientific research. While mixed-mode VLSI multi-neuron chips can support high temporal resolutions by processing continuous analog signals [99, 213, 214], time-stepped digital platforms are bounded by the operating frequency of the global clock [215, 216]. In order to investigate questions about required temporal precision in neural networks, we introduce a novel programming framework for SpiNNaker [120], a digital parallel architecture oriented to the simulation of large scale models of neural tissue. The approach introduced in this work leverages the event-driven nature of the platform to perform simulations with increased temporal resolution. We introduce a new collection of tools (spike sources and monitors, neural and plasticity models) oriented to sub-millisecond event-driven simulations and characterize the temporal behavior of the platform at different levels.

The chapter is organized as follows: Section 5.2 describes the hardware and software architecture of SpiNNaker and its current limitations. Section 5.3 introduces our novel programming framework and the components provided for sub-millisecond simulation. Section 5.4 reports the time characterization of the platform using a new method for measuring latencies. It also present results from two example network models (sound localization and learning of temporal patterns) which require sub-millisecond precision. The networks are implemented in real-time on a 48-chip SpiNNaker board using the novel set of tools presented in this work. Finally the Conclusion section summarizes the key temporal aspects of the software and models introduced in this chapter.

5.2 The SpiNNaker platform

This section describes the hardware and software aspects of the SpiNNaker platform and the current limitations of the software implementation related to time resolution.



(a) The SpiNNaker chip (b) A 48-node SpiNNaker board (SpiNN-4)

Figure 5.1: System overview: (a) shows a plot of the SpiNNaker chip die, while (b) shows the largest prototype which consists of 48 SpiNNaker chips

5.2.1 Hardware

The SpiNNaker chip is an application specific integrated circuit (ASIC) designed to realize large-scale simulations of heterogeneous models of spiking neural networks in biological real-time [120]. Each SpiNNaker chip, Fig. 5.1(a), comprises 18 identical ARM968 cores each with its own local tightly-coupled memory (TCM) for storing data (64 kilobytes) and instructions (32 kilobytes). All cores have access to a shared off-die 128 megabytes SDRAM, where the relevant synaptic information is stored, through a self-timed system network-on-chip (NoC).

At the center of the chip lies a packet-switched multi-cast (MC) router [217] responsible for communicating the spikes to the local cores or to neighbouring chips through 6 asynchronous bi-directional links. The router is capable of handling one-to-many communications efficiently, while its novel interconnection fabric allows it to cope with very large numbers of very small packets. Spikes are transmitted as 40 or 72 bit MC packets implementing the Address-Event Representation (AER) [7] scheme, where the information transmitted is the address of the firing neuron. Each packet consists of an 8 bit packet header, a 32 bit routing key identifying the neuron that fired and an optional 32 bit payload which is not normally used for neural applications [217]. Every core within a SpiNNaker chip includes a communications controller which is responsible for generating and receiving packets to and from the router through an asynchronous communications NoC.

By combining multiple SpiNNaker chips together larger systems are formed. The SpiNNaker board with 48 chips (SpiNN-4), Fig. 5.1(b), is the largest prototype system available to-date and is currently being used as the building block for forming larger SpiNNaker machines. The SpiNN-4 board has 864 ARM9 cores, 768 of which can be used for neural applications, while 1 core per chip is dedicated for monitoring purposes and an additional one for fault-tollerant purposes [218]. Additionally, there are 3 Xilinx Spartan-6 field programmable gate arrays (FPGA) chips that are used for inter-board communication purposes through the 6 high-speed SATA links. A previous study [219] demonstrated that a SpiNN-4 board is capable of handling up to a quarter of a million neurons (with millisecond update), with millions of current-based exponential synapses generating an activity of over a billion synaptic events per second, while each chip dissipating less than 1 Watt. The final SpiNNaker machine will utilize approximately 1,000 SpiNN-4 boards and it aims at simulating a billion neurons with trillions of synapses in biological real-time.

5.2.2 Software

The SpiNNaker software can be divided into two parts, the software running on the chips and on the host. Each SpiNNaker chip runs an event-based Application Run-time Kernel (SARK) that has two threads, the scheduler and the dispatcher. The scheduler is responsible for queuing tasks based on a user-defined priority, while the dispatcher de-queues and executes them starting with the highest-priority task. Tasks with priorities set to minus one are pre-emptive, zero task priorities are non-queueable and are executed directly from the scheduler, while tasks with priorities set to one and above are queueable [173].

The SpiNNaker application programming interface (API) is built on top of SARK and allows users to write sequential C code to describe event-based neuron and synapse models by assigning callback functions that respond to particular system events. Some example events are:

- **Timer Event:** A user-defined periodic event, usually set to 1 ms, which is used to solve the neural equations and update the synaptic currents.
- **Packet Received Event:** An event is triggered every time a core receives a spike (MC packet). It initiates a Direct Memory Access (DMA) transfer in order to fetch the pre-synaptic information from the SDRAM to the local memory. This DMA operation is autonomous, the ARM core may handle pending events or enter into a power-saving sleep mode.
- **DMA Done Event:** This event is generated by the DMA controller to inform the core that a DMA transfer has been completed. Each synaptic weight and conductance delay gets updated.

If there are no pending tasks the cores enter a low-power "sleep" mode.

On the host side PyNN [220], a high-level simulator-independent neural specification language, is used that allows users to described neural topologies and parameters using abstractions such as *populations* and *projections*. A tool named partition and configuration management (PACMAN) [221] is responsible for mapping a PyNN description to a SpiNNaker machine based on the available resources, generating and uploading the relevant binary files, initiating a simulation and fetching the results to the host for further analysis.

5.2.3 Limitations of the current implementation

In the SpiNNaker software available at the time of this work, neural models are implemented in a time-driven fashion. These models use the **Timer event** (see Section 5.2.2) to periodically update the state of the simulated neurons with a given timestep. Parallel to that update process, incoming spikes to the implemented neural population are processed through the **Packet Received Event** (see Section 5.2.2). This event looks up the different synaptic weights and delays relative to each connection. When the synaptic delay has been retrieved, the future contribution of the spike to the membrane potential of a given neuron is stored in its associated Post-Synaptic Potential buffer (PSP buffer). To implement the actual delay, the PSP buffers are ring buffers comprising one cell per simulation timestep. The periodic update process can then read this value at the timestep they need to be applied. This process is represented in Figure 5.2.

This implementation implies a trade-off between time resolution and memory usage. The memory space required by the PSP buffers is proportional to :

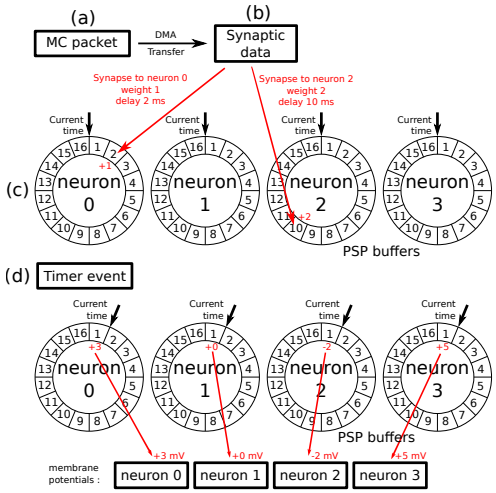


Figure 5.2: Working principle of the PSP buffers: (a) a MC packet is received, carrying an incoming spike. It triggers a DMA transfer. (b) As a result, the DMA Done event is triggered when the synaptic data associated with the spike is retrieved from SDRAM. (c) Synapses related to this spike are processed and their contributions are stored in the PSP buffers of the targeted neurons in the cell of the buffer corresponding to the synaptic delays. (d) When the next Timer Event is issued, the update process reads the contributions of previously received spikes for the current timestep and adds them to the neurons' membrane potential after updating its value (according to the used neural model).

- the number of neurons simulated by the core (one buffer per neuron),
- the maximum synaptic delay allowed
- the resolution of the timestep

The standard implementation uses a maximum delay of 16 ms with a time resolution of 1 ms to simulate 100 neurons per core.

These values have been chosen taking into account spike propagation delays and the memory footprints of the synaptic buffers. Spike propagation in large SpiNNaker machines is guaranteed to happen within 1 millisecond. Original time-driven neural models on SpiNNaker therefore have a 1 millisecond time resolution.

If we want to introduce a time resolution of 1 μs and to keep the same maximum delay, the memory requirements of the PSP buffers increase a thousand times. Moreover, increasing the time resolution means that the periodic update process will be executed more often and thus will have less time to update the state of the population. Going from a millisecond to a microsecond resolution will reduce the time available for state update by a factor of 1000.

It is possible to consider these trade-offs but this results in an dramatic decrease of the number of neurons which can be simulated on a core, going from 100 neurons to only a handful of neurons. Implementing huge neural networks with microseconds resolution is thus not practical with the current available implementation.

5.3 Going beyond the millisecond

This section describes the changes to the standard SpiNNaker software package we implemented to allow simulation with microsecond precision. The changes to the base tools are first discussed, then new neural models are introduced.

5.3.1 Tools and support

5.3.1.1 SpiNNaker API

Applications written for the SpiNNaker cores rely on an Application Programming Interface (API) provided by SpiNNaker's builders. This API provides a set of support functions and a framework allowing the use of the different resources offered by the hardware platform.

Firstly, measuring time with microsecond resolution had to be added to the API which normally measures time with a resolution corresponding to the timestep of the simulation (1 ms by default). This can be easily implemented by using the more precise value of the hardware counter already used by the API to deduce the number of microseconds which have elapsed since the last simulation timestep. This allows precise time measurements without an extra cost in resources.

Secondly, all SpiNNaker events related to timings are based on the simulation clock and thus inherit the timestep resolution. To solve this problem, we use the second hardware timer available in the SpiNNaker cores to generate interrupts with a user settable number of microseconds delay. This allows the scheduling of events which will be triggered with microsecond resolution using the existing framework.

5.3.1.2 Monitoring spikes

To record spikes from a neural simulation in SpiNNaker, a monitoring core runs a special application which collects spikes from the populations selected for recording by the user and sends them to a host computer, in charge of collecting simulation results, while the simulation is running. This is done using a special type of packet which can carry a payload of 256 bytes.

The current implementation of this monitoring application sends a packet of spikes every millisecond. The timestamps of the spikes are indicated in the header of the packet which can then carry a total of 64 spikes. The same scheme cannot be used for microsecond resolution: it would require sending up to a packet every microsecond, potentially containing only one spike. This would dramatically increase the overhead due to the packet headers and would require the emission of too many packets.

To overcome this problem, we implemented a new monitoring application. This core now stores incoming events as a pair of values: (key, timestamp). The key corresponds to the ID of the neuron which spiked and the timestamp indicates the time, in microseconds, when this spike has been received. These pairs are stored in buffers of 256 bytes (32 spikes), which are sent to the shared SDRAM of the chip when full, using DMA. Another process then reads these buffers from SDRAM and sends packet containing the events to the host computer when needed.

This allows more events to be recorded per millisecond than the previous implementation, while allowing microsecond resolution in their time of arrival. Moreover, the events can be stored in SDRAM quicker than they are sent to the host computer, allowing to increase the total amount of data which can be acquired during the simulation (The user will just have to wait for the data to be completely transferred to the host computer after the end of the simulation). This scheme of

pairing addresses and timestamps is also used in other programs such as the jAER program [222] and thus facilitates integration with such tools for real-time software implementation.

5.3.1.3 Generating input spikes

A dedicated application exists to generate input spike trains which appear to come for another population of neurons. The application provided in the standard SpiNNaker implementation uses a particular format to store this information. Spikes are stored in bitmaps. For every simulation timestep, a chunk of memory is read. In this memory portion, each bit codes for one neuron of the source population. If the bit is set to 1, this particular neuron has to produce a spike in the current timestep. If the bit is set to 0, it has to remain silent.

Considering the usual sparseness of spike data, it would be really inefficient to use the same scheme with microsecond resolution. This would require an enormous amount of memory directly proportional to the number of microseconds in the simulation. To improve this process, we implemented a new spike source application. This application represents spikes as pairs of values: (ID, timestamp). The information is stored in SDRAM by the host computer before the start of the simulation and read via DMA throughout the simulation. When a spike is produced, the application reads the time when the next spike should be produced. This allows us to compute the wait time before producing this next spike. An event is then scheduled in the API to happen after this delay. This process goes on during the whole simulation time until the end of the spike data. A pipelined process using a double buffer technique allows one to read the spike data from memory using DMA without adding delays in the replay process sending them to the other neural models of the simulation.

5.3.2 Neural models

5.3.2.1 Dendritic delays

As we can see from the standard implementation provided with SpiNNaker, managing synaptic delays when simulating neurons can lead to an important memory usage. To solve this problem, we chose to implement these delays independently from the rest of the neural simulation.

One dendritic delay core implements one particular delay value. When a packet containing a spike is received, it is stored in a ring buffer in DTCM (local memory of the core). Then, a second process schedules events to dispatch these spikes after the given delay of the core has elapsed. This allows very compact and efficient code because events are output in their order of arrival.

Because there are a large number of cores available in a typical SpiNNaker machine, using one core per delay value is not troublesome. Moreover, one could configure a network where a spike goes several times through the same delay core to implement multiples of a base delay: if one core implements a delay of 100 μs, it can be used to realize a delay of 300 μs by routing events three times through the core before delivering the spike to its target neuron.

5.3.2.2 Synchrony detectors

Detecting temporal coincidence between two spikes is a widely used feature in spiking neural networks [223, 188]. As a consequence we decided to implement a dedicated core for this task instead of using standard integrate and fire neurons which would introduce an unnecessary overhead.

Each neuron simulated by this core has two types of synaptic input and a time window. When an incoming spike is received on one input, the core will output a spike if another spike was received on its second input in the given time window. We added a refractory period to this process to limit the maximum firing rate of the neurons if required.

5.3.2.3 Leaky integrate and fire neurons

While an *ad hoc* solution can be introduced in the case of synchrony detection, for the rest of the network simulation we need to simulate standard neural models. This is done through a new core which implements leaky integrate and fire models with exponential kernels. These neurons are completely event-driven and have microsecond resolution.

As in the standard models, incoming packets carrying input spikes are received and queued in memory for processing (this process has to be done as soon as possible to ensure no back-pressure signal is propagated to the SpiNNaker router by not reading packets, as described in the next Section). In addition to this standard processing, we also timestamp the spike on arrival to be sure to it is processed correctly even if events are queued for a variable amount of time. When such a spike is processed, a DMA call is issued to get the associated synaptic information from SDRAM.

Upon receiving the synaptic data from memory, the membrane potential of the post-synaptic neuron is updated with the exponential kernel and the spike's contribution is added to this potential. Its value is then tested for an output spike which is immediately produced if necessary.

It is worth noting that an output spike can only be produced at the time of an input spike which is compatible with an exponential kernel and with the absence of synapse dynamics (the contribution of a spike is an instantaneous addition/subtraction to the membrane potential according to the synaptic weight).

To optimize the process, the exponential kernels are precomputed by the core before the beginning of the simulation via two look-up tables. When we update the membrane potential, we need to compute the kernel decay over the time Δ_t which has elapsed from its last update:

$$e^{-\Delta_t/\tau} = e^{-(\Delta_t^{\text{ms}}+\Delta_t^{\mu\text{s}})/\tau} = e^{-\Delta_t^{\text{ms}}/\tau} e^{-\Delta_t^{\mu\text{s}}/\tau}, \tag{5.1}$$

where Δ_t^{ms} is the maximum multiple of milliseconds contained in Δ_t, $\Delta_t^{\mu\text{s}}$ is the remaining number of microseconds and τ is the time constant of the neuron. To reduce the memory footprint required by the look-up table, we compute one table with microsecond resolution for the $e^{-\Delta_t^{\mu\text{s}}/\tau}$ part over a timespan of 1 ms and a second one with millisecond resolution for the $e^{-\Delta_t^{\text{ms}}/\tau}$ part over a timespan which can be configured in the code depending on the available local memory and time constant τ.

5.3.2.4 Plasticity

Implementing plasticity on the SpiNNaker system is not a trivial task, due to its peculiar set of constraints and architectural characteristics. Rules that rely on spike timing can be triggered by the arrival of a pre-synaptic spike, inducing depotentiation, or the emission of a post-synaptic spike signaling depotentiation. This is for instance the case for spike timing-dependent plasticity (STDP) [126]. On SpiNNaker, weights from SDRAM are only available in the local memory of an ARM core upon the receipt of a MC packet. This triggers a lookup in memory fetching all the weights associated with the incoming spike. Weights are therefore available in memory only when a packet is received. Due to the nature of how delays are normally implemented on the

SpiNNaker architecture (see Section 5.2.3), this time does not correspond to the time the spike arrives at the post-synaptic neuron, as the delay is reintroduced post-synaptically. Moreover, when a post-synaptic spike is produced weights are in SDRAM, with no local pointer to them; retrieving them selectively would be difficult as they are indexed by pre-synaptic neuron, hence scattered when considering post, leading to unoptimized memory transfers.

The deferred event-drive model (DED) [177] was introduced to circumvent these problems. The approach consists in gathering information about spike timing and deferring plasticity into the future, once all the information required is available. In our implementation, we decided to use the voltage-gated STDP variant introduced by [142]. This rule is particularly appealing for neuromorphic implementation (see for example [186]) because it is only triggered on the arrival of a pre-synaptic spike, solving the first of the two problems associated with plasticity implementation. Our novel delay cores take care of solving the second problem, the reintroduction of the delay at the post-synaptic end. By using these cores, the time of arrival of a MC packet corresponds to the time when a spike needs to be computed at the post-synaptic end. This rule depends on a post-synaptic trace $C(t)$ representing the calcium concentration and which evolves accordingly to the firing activity of the neuron:

$$\frac{dC(t)}{dt} = -\frac{C(t)}{\tau_C} + J_C \sum_i \delta(t - t_i) \quad , \tag{5.2}$$

where t_i are the post-synaptic spike times. $C(t)$ triggers potentiation and depression as follows: if $C(t)$ is in an interval $[\theta^h_{down}, \theta^h_{up}]$ potentiation is triggered; otherwise if $C(t)$ is between $[\theta^l_{down}, \theta^l_{up}]$ depression is triggered. This variable is computed using an exponential LUT in a similar way as done with the membrane potential. The plasticity rule depends also on the post-synaptic membrane depolarization $V(t)$ according to a threshold value θ_V, sampled at the time of arrival of a pre-synaptic spike (t_{pre}). Weight dynamics $w(t)$ can then be summarized as follows:

$$w = w + a \quad \text{if} \quad V(t_{pre}) > \theta_V \quad \& \quad \theta^h_{down} \leq C(t_{pre}) < \theta^h_{up} \tag{5.3}$$

$$w = w - b \quad \text{if} \quad V(t_{pre}) \leq \theta_V \quad \& \quad \theta^l_{down} \leq C(t_{pre}) < \theta^l_{up}, \tag{5.4}$$

where a and b represent the constant weight increase and decrease values respectively.

The plasticity rule provides relaxation towards two stable states, if none of the conditions in (5.3) and (5.4) are fulfilled. The weight w drifts linearly with rate α towards w_{max} if its value is greater than a threshold θ_W; conversely it drifts linearly towards w_{min} with rate β leading to the additional dynamics:

$$\frac{dw(t)}{dt} = \alpha \qquad \text{if} \quad w(t) > \theta_W \tag{5.5}$$

$$\frac{dw(t)}{dt} = -\beta \qquad \text{if} \quad w(t) \leq \theta_W \tag{5.6}$$

Using this system we can efficiently compute weight updates upon the arrival of a pre-synaptic spike, as all the information required by the algorithm are locally available in the neural core.

5.4 Results

This section presents the time characterization of the platform and of the novel software infrastructure introduced. The section concludes by showing two simple experiments which use the new

features: a binaural model for sound localization and a plastic model capable of learning precise temporal relationships in spike trains.

5.4.1 Intra- and Inter-Chip MC Packet Latencies

The MC router is responsible for communicating the spikes to its internal cores or to other chips through six asynchronous bi-directional links. Its pipelined implementation enables it to route one packet per clock cycle to all or a desired number of output links in an uncongested network. If any of the output links is busy, the router will retry to route the packet at every clock cycle until it reaches a predefined number of clock cycles after which it will attempt emergency routing via the link which is rotated one link clockwise from the blocked link (not applicable for destinations internal to a SpiNNaker chip). Similarly, if the emergency route fails the router will retry emergency routing at every clock cycle until it reaches a second user-defined number of cycles when it finally drops the packet.

This section describes a series of experiments conducted in order to investigate the intra- and inter-chip MC packet latencies as a function of the router's waiting time and synthetic traffic going through a link. For these experiments, a parametrised software was developed using the SpiNNaker API. The packet received callback priorities were set to minus one (pre-emptive) in order to ensure that packets were cleared from the communications controller immediately upon receipt. The timer tick callback priority, which was used by the cores for terminating the simulation after 60 seconds, was set to zero (non queueable priority). Finally, the priority of the callback function developed to generate MC packets was set to two (lowest queueable priority). The processor clocks were set to 200 MHz, routers and system buses to 100 MHz, while the off-die memory clocks to 133 MHz.

5.4.1.1 Intra-chip MC packet latency

The first experiment was aimed at demonstrating the core-to-core packet latency within a SpiNNaker chip as a function of a congested internal link and the wait time of the router before dropping a packet. Congested link means that a core has received more packets than it can process on time and a back-pressure signal will propagate to the router through that link. For this experiment, 17 cores were used. One core was dedicated to measuring the core-to-core MC packet latency within a SpiNNaker chip. By sending a packet to the router every 500 milliseconds using the "timer tick" callback function and receiving it back. The second hardware timer, available on each core, was used to count the clock cycles between sending and receiving the MC packet (with nanosecond resolution). Each of the remaining 15 cores would generate approximately 1.7 million packets per second, which would be routed to one particular consumer core (C) whose sole task was to count the received packets, Figure 5.3. The logged MC packet latencies during the simulation, the values of the software counters and additional diagnostic information from the router were uploaded to the SDRAM when the simulation was over and fetched by the host for further analysis.

Figure 5.4 shows the mean and standard deviation of the intra-chip MC packet round-trip delay time (RTD) as a function of the total number of MC packets per second the router has issued to the consumer core (C) and for various router wait times. What can be observed from this figure is that in an uncongested network, the intra-chip round-trip delay time is constant at 0.825 μs, Figure 5.4(a). Within this time is included the software overhead of the SpiNNaker API required to write the MC packet to the communication controller, the time needed for the packet to traverse through

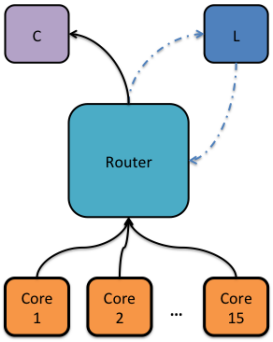


Figure 5.3: Block diagram showing the topology used to measure the intra-chip packet latency as a function of a congested link and the router's waiting time before dropping a packet. Cores 1 to 15 were used to generate synthetic MC packets and the router would redirect these packets to a consumer core (C). An additional core was used to measure the MC packet latency (L) by sending a packet to the router periodically and receiving it back. A hardware timer was used to measure the time passed from sending a MC packet to receiving it back.

the internal link to the router, the time required for the packet to go through the router and again through the internal link to the communication controller of the target core and finally the software overhead of receiving the packet from the communication controller. The aforementioned times can be expressed as:

$$t_{\text{RTD}}^{\text{Intra}} = t_{\text{Send}}^{SW} + 2 \cdot t_{\text{Link}}^{\text{local}} + t_R + t_{\text{Receive}}^{SW} \quad , \tag{5.7}$$

where t_{Send}^{SW} is the software overhead of the API required to write the key of a MC packet to the communication controller, t_{Receive}^{SW} is the time passed from handling the interrupt raised by the communication controller to branching to the callback function assigned to handle the packet received events, t_R is the time required by the router to process a single MC packet, and finally $t_{\text{Link}}^{\text{local}}$ is the time a single MC packet needs to go through the local links.

The mean t_{Send}^{SW} and t_{Receive}^{SW} times (averaged over 4 trials) were found to be 0.415 μs and 0.13 μs respectively, by utilising the second hardware timer. Assuming that the time consumed by a MC packet to traverse through the local links is much smaller than the time spent by the router to process a packet, the $t_{\text{Link}}^{\text{local}}$ can be ignored. Solving equation 5.7 for t_R, the time the router requires to process a single MC packet is 0.28 μs.

As soon as congestion occurs, which for this experiment happens when the consumer core (C) receives more than 3.6 million packets per second, the communication controller of the consumer core (C) starts adding back-pressure on the router, which attempts to resend the packet at every clock cycle until it reaches a predefined number of cycles (240 default) at which point the packet is finally dropped, Figure 5.4(b). This back-pressure signal propagates back along the pipeline and the router stops receiving new packets until back pressure has been released [217]. As a consequence, the MC packet latency increases and the hardware buffers of the communication controllers of the cores generating the MC packets are not emptied; this explains why the total number of generated packets plateaus, as seen in Figure 5.4 (c), while failed packets increase (software buffer full), see Figure 5.4 (d). For the router's default waiting time (240 cycles) and for waiting 60 cycles no packets were dropped in any of the trials but the worst-case round-trip

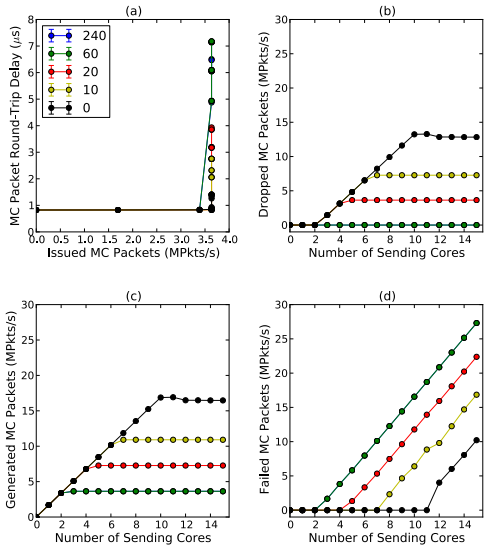


Figure 5.4: (a) shows the mean intra-chip MC packet round-trip delay time as a function of the total number of packets per second the router issued to the consumer core (C), for different router wait times (default value is 240 cycles). (b) shows the total number of packets per second the router dropped as a function of the number of cores participating in the simulation and for different router wait times. (c) shows the total number of packets per second that were successfully generated by the communication controller as a function of the number of cores participating in the experiment and for different wait times. Finally, (d) shows the total number of packets per second the communications controller was not able to send to the router due to the back-pressure of a congested link, for different wait times. Each core attempted to inject 1.7 million packets per second to the consumer core through the router.

delay time went up to 6.5 μs. For router wait times of 20 cycles and below, the worst-case round-trip delay time drops below 4 μs but the total number of dropped packets per second increases dramatically. This trade-off between intra-chip MC packet latency, packets being dropped or not being sent to the router at all, requires further investigation as it depends on the requirements of a particular application.

5.4.1.2 Inter-chip MC packet latency

The router of a SpiNNaker chip can communicate packets to neighbouring chips through six self-timed bi-directional links. The average bandwidth (transmit/receive) of each link is 6 million packets per second (240 gigabits per second) and this may vary with the temperature, voltage or silicon properties. An experiment was conducted to determine the inter-chip RTD of a MC packet transmitted through one of the 6 bi-directional links as a function of the link's outgoing and incoming traffic. For this experiment a core (L1) generates a MC packet every 500 milliseconds and the router routes it to a neighbouring chip through one of the six bi-directional self-timed links. Upon receiving the packet the second router would route it to a particular core (L2), whose sole task was to change the key of the packet and retransmit it back to the router which had an

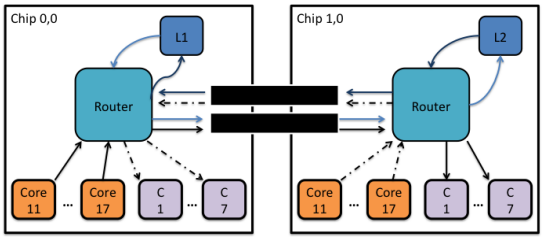


Figure 5.5: Block diagram showing the topology used to measure the inter-chip MC packet latency as a function of synthetic traffic going through a link.

appropriate routing entry to route it back to the originating core, Figure 5.5.

Seven cores on each chip were used to generate packets which were routed to seven consumer cores (C) on the adjacent chip following a one-to-one mapping. This way the total number of packets per second a consumer core receives remains below 3.6 million packets per second, (which is the maximum number of packets a core can receive) ensuring that no additional pressure is added to the routers.

The generated packets were controlled by an inter-packet interval (IPI) parameter, which is a delay in microseconds before transmitting the next MC packet. Results are presented as the percentage of the utilisation of the incoming and outgoing packets going through a link per second, with 100% utilisation meaning 6 million packets per second.

The mean and standard deviation of the round-trip delay times of MC packets are presented in Figure 5.6. When there is no traffic the round-trip delay time of a MC packet is 2.535 μs. Within this time is embedded the time required for the packet to go through each router twice, twice through the external link, and also two software processing overheads of sending and receiving the packet back to the router. This can be expressed as:

$$\begin{aligned} t_{\text{RTD}}^{\text{Inter}} &= 2 \cdot t_{\text{Send}}^{SW} + 4 \cdot t_{\text{Link}}^{\text{local}} + 4 \cdot t_R \\ &\quad + 2 \cdot t_{\text{Receive}}^{SW} + 2 \cdot t_{\text{Link}}^{\text{external}} \end{aligned} \tag{5.8}$$

where $t_{\text{Link}}^{\text{external}}$ is the time a MC packet requires to traverse through an external link to a neighbouring chip. Solving for an RTD time of 2.535 μs and by using the results of equation 5.7 for t_R and by ignoring the time of $t_{\text{Link}}^{\text{local}}$, the time a MC packet needs to go through an external bi-directional link is 0.1625 μs.

For a link utilisation of 60%, for both outgoing and incoming traffic, there is a 3% increase in the RTD time. When both the incoming and outgoing link utilisation reaches 80% a very small number of packets were dropped as both routers attempted to reroute the packets for the default wait times (240 cycles), hence the dramatic increase in the RTD times. Results are summarised in Table 5.1.

5.4.2 Time characterization

To test latencies in the system we build and simulate a very simple network composed of three populations:

- a spike source, producing spikes with microsecond resolution,

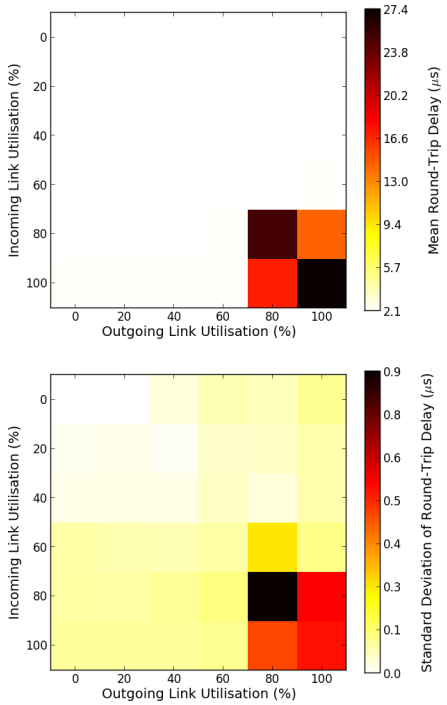


Figure 5.6: Mean and standard deviation of round-trip delay times of MC packets as a function of the percentage of a link's utilisation. Top figure shows the mean round-trip delay time, while bottom figure shows the standard deviation in microseconds. 100% utilisation means 6 million outgoing/incoming packets per seconds going through a bi-directional link.

Table 5.1: Experimental results of the SpiNNaker latencies.

Parameters	Values	Units
t_{Send}^{SW}	0.415	μs
t_{Receive}^{SW}	0.13	μs
t_R	0.28	μs
$t_{\text{Link}}^{\text{external}}$	0.1625	μs

- a dendritic delay population which delays spikes with a tunable delay,

- a population of integrate and fire neurons in which synaptic plasticity is enabled or not in the implementation.

We add to the global architecture the monitoring application which records spikes from these 3 populations. All these applications run on the same SpiNNaker chip. From here, we can characterize timings and latencies by either looking at timestamps generated by the monitoring core which gives us a common clock for all of the activity on the chip or by keeping track of the timings directly in the different cores by adding debug instructions to their processing.

We start by characterizing the spike source application. By adding debug instructions in the code, we can output the time at which spikes would be sent without this debugging overhead and compare them to what is asked from the core. Our measurements show that every spike is reliably emitted with a 2 μs delay after the timestamp set in the simulation script in conditions where no delays are introduced by congestion due to too much activity in the chip. Looking at the same events in the output of the monitoring application, we can see that they are consistently timestamped with a 3 μs. The 1 μs difference between the two numbers is due to the overhead of the API when sending a spike, that is, the time it needs to travel from one core of the chip to another one (which has been characterized in the previous subsection) and the time needed by the receiving core to timestamp the spike. From the results of the previous subsection (5.4.1.1), we can consider than under normal operational conditions (no excessive load of events in the system), this time will be constant for every population on the same chip because they use the same code to send packets to the system. Thus, from now on, we can use the timestamps from the monitoring application to know when spikes are emitted by each core recorded during our simulations.

We then look at the dendritic delay population. This population directly receives input from the spike source and the delayed output is recorded by the monitoring application. By comparing the timestamps of the spikes received from the spike source and the ones received from the dendritic delay core, we can compute the actual delay introduced by this population. By removing its set delay, we obtain the overhead introduced by the implementation. For different delays and input spikes, we reliably get an overhead of 2 μs which can be compensated when specifying a desired delay in a simulation.

This delay population is then connected to a population of integrate and fire neurons implementing plasticity as described in the previous section. The connection between these two populations is an all-to-all connection. This allows us to vary the size of the synaptic data fetched from memory, which directly depends on the number of post-synaptic neurons associated to each pre-synaptic one. This enables us to vary the amount of processing each incoming spike requires (more post-synaptic neurons means more neurons to update when receiving a spike), thus allowing easy characterization of the following latencies:

- the initial latency, corresponding to the time required to receive the spike and fetch the synaptic data from memory. This time has been measured to be 4 μs,

- the time required to update each post-synaptic neuron targeted by the incoming spike. This time has been measured to be 1.6 μs,

- the time required to send a spike which has been measured to be 0.4 μs.

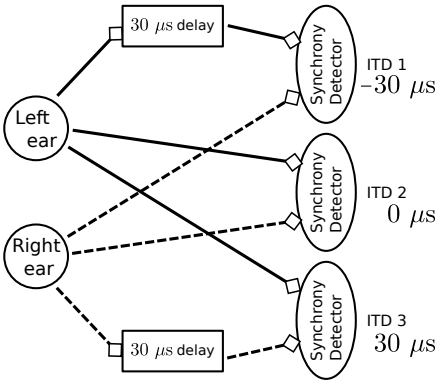


Figure 5.7: Model used to detect sub-millisecond spike synchrony for sound localization. The model consists of three synchrony detectors where each is activated by a different input interaural time difference (ITD). To achieve this result, each detector is directly connected to one ear whereas the second input comes from a dendritic delay population. For positive ITDs, spikes from the right ear are delayed. For negative ITDs, spikes from the left ear are delayed. In this experiment, spike trains from the two ears are simulated for three sound sources localized at positions producing expected ITDs.

These timings allow to compute the required update time T_u required by a spike for a given network topology with the following formula:

$$T_u = 4 + 2 \cdot N_{\text{max}}^{\text{post}} \quad \mu\text{s}, \tag{5.9}$$

in the worst case scenario where every pre-synaptic spike produce a spike in each one of its post-synaptic neurons and where $N_{\text{max}}^{\text{post}}$ is the maximum number of post-synaptic neurons a pre-synaptic one connects to.

If we remove the plasticity computation from the model, the initial latency and the time needed to produce a spike stay the same. The only modification can be found, as expected, in the time needed to process a spike which drops to 0.5 μs.

5.4.3 Detecting sub-millisecond spike synchrony in a model of sound localization

To test the dendritic delay and synchrony detector models, we simulate a standard network used for sound localization. This model is presented in Figure 5.7 and results are presented in Figure 5.8. For each ear, we consider a population of neurons representing 10 different frequency channels (Panel (a) in Fig. 5.8). We start by generating spike trains with an interspike interval (ISI) of 100 μs for each of these channels and we feed them in the right ear (red dots). Then, this simulated sound is shifted in time according to the input interaural time differences (ITDs) corresponding to values compatible with human hearing: -30 (phase (1)), 0 (phase (2)) and 30 μs (phase (3)) to generate the input spikes for the left ear (blue dots). Some noise is then added independently to spikes from each ear and each channel by jittering each spike randomly between -5 and 5 μs to get the actual input presented in Figure 5.8(a). Each ear is then input in delay lines and synchrony detectors such as to detect the corresponding ITDs, synchrony detectors are, because of their associated delay lines, centered around -30, 0 and 30 μs with a window of 15 μs. These detectors are color coded in Fig. 5.8(b) with detectors for ITDs -30, 0 and 30 μs respectively

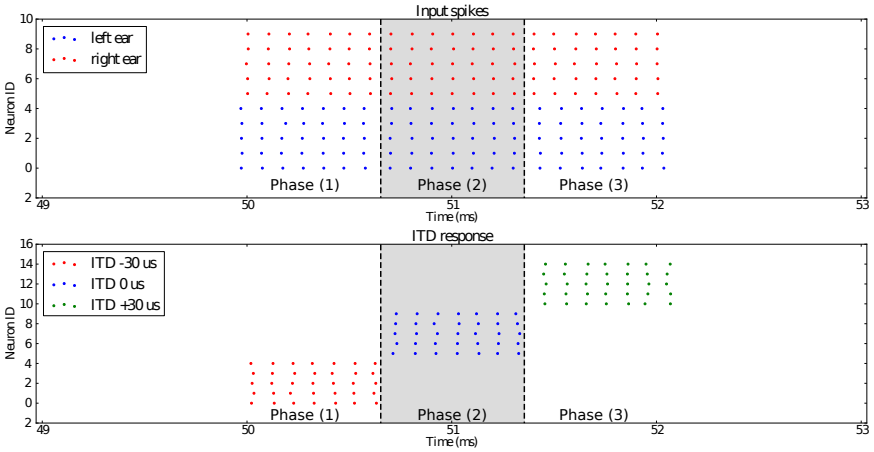


Figure 5.8: Top plot (a) shows the input spikes to the system. It is comprised of the spikes (5 channels, red dots) from the right ear and the spikes (5 channels, blue dots) from the left ear. They correspond to a sound source positioned at ITD -30 μs for the first third (phase (1)) of the input stimulus, then ITD 0 μs for the second third (pahse (2)) and 30 μs for the last part (phase (3)). Bottom plot (b) presents the outputs of the three synchrony detectors of the network configured to respond to ITDs -30 μs (red), 0 μs (blue) and 30 μs (green). We can see that for each channel, the detector with its preset ITD fired correctly in each phase of the experiment.

corresponding to red, blue and green dots. We can see that the different input ITDs are correctly extracted by the architecture for each phase of the input patern.

5.4.4 Learning temporal patterns with sub-millisecond precision

The model presented in the previous section describes how synchrony detection can be exploited through delay lines to localize the source of a sound. In this section, we use the model introduced by [188] to learn spatio-temporal patterns with sub-millisecond precision. The model shown in Figure 5.9(a), can be described as tonotopically organized channels, as it is in the auditory system. Each frequency channel consists of three neurons (A, B_1, B_2) interconnected through delay lines C. Within a channel, neuron A, representing inputs from a channel of a silicon cochlea for e.g. [224], connects to both neurons B_1 and B_2 through excitatory synapses while neuron B_1 is connected to B_2 through an inhibitory synapse and neuron B_2 receives excitatory plastic input connections from the neighboring B_1 neurons. These connections are mediated by the delay populations, implemented with the model described in Section 5.3, so as to have a delay proportional to the tonotopic distance between two channels. This delay line is the feature that enables learning of temporal patterns through coincidence detection.

To test our newly introduced SpiNNaker infrastructure for microsecond precision and delays, we reproduce the results published by [144] on the implementation of the model on a plastic neuromorphic analog VLSI multi-neuron chip. We present the same three spiking patterns as in the paper: a forward, a backward and a forked frequency sweep, activating all channels in rapid succession but with different time dynamics. Each channel is activated with 10 spikes with an ISI of 250μs and each channel is activated with a delay of 500μs after the previous channel. Each frequency sweep has been repeated 20 times, resulting in the connectivity matrices presented in

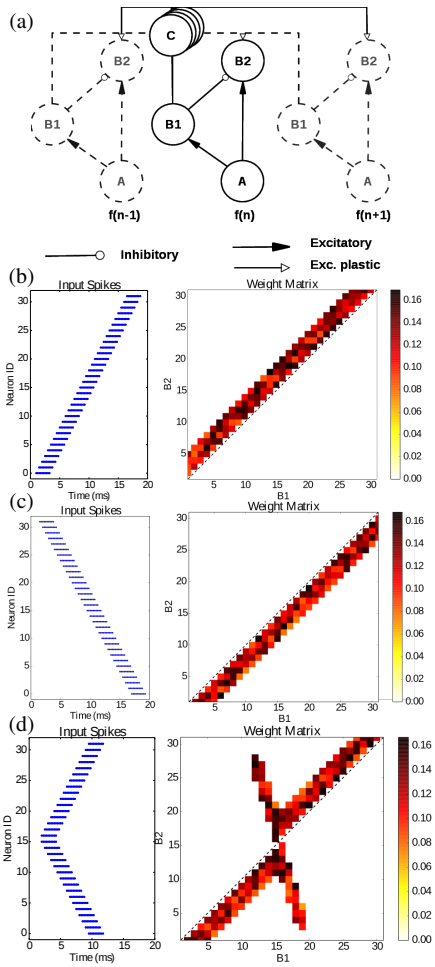


Figure 5.9: Learning sub-millisecond time patterns: (a) network structure: input spikes are produced by neurons A, which represent frequency channels of a simulated silicon cochlea; they drive neurons B_1 and B_2 with excitatory synapses; B_1 is connected to the B_2 neuron of its own channel with an inhibitory synapse, while it is connected to the other channels through the delay populations C with plastic connections. Delays are proportional to the tonotopic distance between channels. (b) Input raster plot (left) and final weight distribution (right) when stimulating the network with a forward frequency sweep, activating all channels in rapid succession. (c) Input raster plot (left) and final weight distribution (right) when stimulating the network with a backward frequency sweep, activating all channels in rapid succession; (d) input raster plot (left) and final weight distribution (right) when stimulating the network with a forked frequency sweep starting from the central channels.

Figure 5.9(b)-(d) which can be compared with Figures 7 and 8 in [144]. In each case training took approximately 4 seconds. The resulting weight matrices, initialized randomly, show how the network was able to learn a precise spatio-temporal pattern through coincidence detection. The learning process potentiates the synapses detecting the temporal features of the presented stimulus thanks to the tonotopic delays lines, converging in an emergent connectivity matrix which tunes the network to the presented stimulus.

The same experiment has also been replicated on SpiNNaker using the framework introduced in [20], but with important differences in the temporal resolution and in the methodology to what is presented here. The temporal resolution of the former work is limited to the millisecond precision because of the structure of the neuron models and plasticity framework used. This new framework allows us to compute plasticity with a time resolution of less than a millisecond, which was not possible with the previous plasticity methods implemented on SpiNNaker.

When comparing our resulting weight matrices to the ones from [144], it can be noted that our results are not impacted by 2 important factors of the hardware system used in the original work: the precision of the synaptic weights and the mismatch between neurons. Since SpiNNaker is a digital platform, the precision of the weights can be changed depending on the application. As a result, our results show a finer scale in the weight matrix. Similarly, SpiNNaker neurons and synapses are not affected by hardware mismatch because of the digital implementation which results in less noisy weight matrics in our implementation.

This work matches the temporal resolution of the experiments in [144]. Regarding the methodology, in this work we use a purely event-driven approach, that is plasticity is computed as soon as a spike is received, leveraging the simplification introduced by the separation of synaptic delays and neural models. This work hence constitutes the first implementation of plasticity on SpiNNaker where the weight update is not deferred into the future.

5.5 Discussion

In this chapter, we showed that the current software package provided with the SpiNNaker platform was insufficient for certain applications requiring temporal resolution below a millisecond. To overcome this limitation, we introduced new software tools and models allowing to go beyond the millisecond barrier and reach microsecond precision.

To assert these new functionalities, we needed information about the timing involved at the level of SpiNNaker's fabric itself. We characterized the timing requirements of the hardware on one hand and of the software API on the other. This allowed us to fully characterize our implementation and provide insights about its computational limits for an user wanting to simulate a given network. We then demonstrated that our newly introduced architecture can simulate networks implementing sub-millisecond tasks and using plasticity by achieving, in real-time, sound localization and sound patterns extraction with realistic spike trains.

It should be noted that these new tools do not change the way in which events are transferred in the global SpiNNaker architecture. They are just making the best of real-time to increase the time resolution of implemented models. Thus, they are fully compatible with the already existing models on SpiNNaker. This means that if a user wants to build a sensory processing neural network in which microsecond resolution is only needed in the early processing stages, he or she can use these microsecond precision models in these stages and then feed their output to standard neural models which will then compute with millisecond resolution for the later stages. This

allows resources to be exploited maximally by tuning the time resolution to the requirements of the running model.

Notably we have introduced learning in our framework as it is a key process in developing precise coincident detectors. Furthermore in the already mentioned owl auditory system, cross-modal interaction of different sensory systems appears to be crucial: visual cues guide the formation of a precise sound localization neural circuit [211, 212]. These studies point at the importance of representing sensory inputs with high temporal resolution. In fact our newly introduced framework is much oriented to exploiting the enhanced temporal properties given by neuromorphic event-driven sensors [8], as silicon retinas [63, 13, 11] and cochleas [224] which can seamlessly be interfaced with SpiNNaker [225, 226]. In this regard we have presented here a platform which offers a wide range of trade-offs in simulating spiking neural networks with different time-scales efficiently, and can be used for cross-modal learning.

Chapter 6

STICK: Spike Time Interval Computational Kernel

The paradigm shift introduced by new event-based algorithms designed to process data from neuromorphic sensors is well suited for implementation on new hardware platforms designed for highly parallel processing. But these platforms still are a remnant of standard machines which were designed to execute a very different kind of algorithms. Re-thinking our approach of processing algorithm is the first step towards a new way of sensing the world but we need to go beyond this step by also re-thinking the platforms we are using to implement these algorithm. Going towards new architectures, free from the current memory bottleneck introduced by the Von Neumann *architecture and natively parellelizing computation is the next logical step. This chapter proposes a possible framework for such a new kind machine.*

Contents

6.1 Introduction

More than 50 years after the first *Von Neumann* single processor, it is becoming more and more evident that this sequential power greedy architecture scales poorly to multiprocessors. Despite the increase of the size of on-chip cache to stay away from RAM and to put the data closer to the processor, major processor manufacturers have run out of solutions to increase performances. The current solutions to use multicore devices and hyperthreading tries to overcome the problem by allowing programs to run in parallel. This parallelism is however limited as hyper-threaded CPUs even if they include extra registers still have only one essential element of most basic CPU features [227].
The quest for a more power efficient alternative has seen a major breakthrough these last years, specially in asynchronous brain like dataflow architectures. Recent endeavours, such as the SyNAPSE DARPA program, led to the development of silicon neuromorphic neural chip technology that allows to build a new kind of computer with similar function, and architecture to the brain. The advantage of these systems is their power efficiency and the scaling of performance with the number of neurons and synapses used. There are currently several available platforms, to cite the most sucessful: IBM TrueNorth [228], Neurogrid [229], SpiNNaker [14], FACETS [230]. These machines seem to be primarly intended to simulate biology, their main application being currently in the field of machine learning and more specifically running deep learning architectures such as deep neural networks, convolutional deep neural networks, deep belief networks and recurrent neural networks. These techniques have shown to be efficient in several fields such as machine vision, speech recognition and natural language processing. Other options exists such as the Neural Engineering Framework (NEF), that has shown to be able to simulate brain functionalities and provides networks that can accomplish visual, cognitive, and motor tasks [231]. NEF intrinsically uses a spike rate-encoded information paradigm and a representation of functions using weighted spiking basis funtions; it thus requires a very large number of neurons to compute simple functions. Other methods synthetize spiking neural networks for computation using Winner-Take-All (WTA) networks [232], or more vision dedicated spiking structures such as convolutional neural networks [233]. Linear Solutions of Higher Dimensional Interlayers networks [234] are another class of approaches that are currently used to derive what is called Extreme Learning Machine (ELM) [235]. Recently the Synaptic Kernel Inverse Method (SKIM) framework [89] has been introduced, it uses multiple synapses to create the required higher dimensionality for learning time sequences. These methods use random nonlinear projections into higher dimensional spaces [236, 237]. They create randomly initialized static weights to connect the input layer to the hidden layer, and then use nonlinear neurons in the hidden layer (which in the case of NEF are usually leaky integrate-and-fire neurons, with a high degree of variability in their population). The linear output layer allows for easy solution of the hidden-to-output layer weights; in NEF this is computed in a single step by pseudoinversion, using singular value decomposition.

Our method goes beyond the classical point of view that neurons transmit information exclusively via modulations of their mean firing rates [238, 239, 240]. There seems to be growing evidence that neurons can generate spike-timing patterns with millisecond temporal precision in [241, 242, 243, 244, 245, 246, 247]. Converging evidence suggests also that neurons in early stages of sensory processing in primary cortical areas (including vision and other modalities) use the millisecond precise time of neural responses to carry information [52, 248, 249, 250, 251].
Our approach will also make use of precisely timed transmission delays. The propagation delay

between any individual pair of neurons is known to be precise and reproducible with a submillisecond precision [252, 253]. Axonal conduction delays in the mammalian neocortex [252, 254, 255] are known to range from 0.1 ms to 44 ms. Finally we will also use the property of biological neurons that states the same presynaptic axon can give rise to synapses with different properties, depending on the type of the postsynaptic target neuron [256, 257, 258].

In this chapter we are interested in deriving a new paradigm for computation using neuron like units and precise timing. The goal is to design micro neural circuits operating in the precise timing domain to perform mathematic operations. We show that when using computation units that have common properties with biological neurons such as precise timing, transmission delays, and synaptic diversity, it becomes possible to derive a Turing complete framework that can compute every known mathematical function using a non Von Neumann architecture. The presented framework allows to derive all mathematical operators whether they are linear or non linear. It also allows relational operations that are essential to develop algorithms. The precise timing framework has a compact representation and it uses a low number of neurons to solve complex equations. Examples will be shown on first and second order differential equations.
The developed methodology is in adequation with scalable neuromorphic architectures that make no distinction between memory and computation. Every synapse of each computational unit of the model simultaneously stores information and uses this information for computation. This contrasts with conventional computers that separate memory and processing thus causing the *von Neumann bottleneck* where most of the computation time is spent in moving information between storage and the central processing unit rather than operating on it [259]. The developed approach is easily scalable and is designed to naturally operate using an event-driven massive parallel communication similar to biological neural networks.

The next section describes the neural model used in this work and the encoding scheme chosen to represent values in the exact timing of spikes. It also describes neural networks implementing elementary operations that can be assembled to implement arbitrary calculus. We then present results of applications of such networks, followed by a discussion on the methods proposed in this work and a conclusion.

6.2 Methods

6.2.1 Neural model

These neuron-like computational units use the following neural model:

$$\begin{cases} \tau_m . \frac{dV}{dt} &= g_e + gate.g_f \\ \frac{dg_e}{dt} &= 0 \\ \tau_f . \frac{dg_f}{dt} &= -g_f \end{cases} \tag{6.1}$$

V is the membrane potential of the neuron. We consider here that there is no leakage of the membrane potential (or that the time constant of this leakage is much slower than all the other time constants considered in this work, in which case it can be neglected). g_e represents a constant input current which can only be changed by synaptic events. g_f represents input synapses with exponential dynamics. These synapses are gated by the $gate$ signal which is triggered by synaptic events. For the experiments presented in this work, we use $\tau_m = 100$s and $\tau_f = 20$ms.

We thus distinguish 4 type of synapses, where w is the weight of the synapse:

- $V-$synapses directly modify the membrane potential value: $V \leftarrow V + w$,
- g_e-synapses directly modify the constant input current: $g_e \leftarrow g_e + w$,
- g_f-synapses directly modify exponential input current: $g_f \leftarrow g_f + w$,
- $gate-$synapses : $w = 1$ activate the exponential synapses by setting $gate \leftarrow 1$; $w = -1$ deactivate the exponential synapses by setting $gate \leftarrow 0$.

All synaptic connections are also defined by a propagation delay between the source and target neurons.

A neuron spikes when its membrane potential reaches a threshold, i.e.:

$$V \geq V_t \tag{6.2}$$

it then emits a spike and is reset by putting back its state variables to:

$$V \leftarrow V_{\text{reset}} \tag{6.3}$$
$$g_e \leftarrow 0 \tag{6.4}$$
$$g_f \leftarrow 0 \tag{6.5}$$
$$gate \leftarrow 0 \tag{6.6}$$

without loss of generality, we will consider $V_t = 10\text{mV}$ and $V_{\text{reset}} = 0$ to simplify the following equations.

In the following subsections, we use the following notations. T_{syn} is the propagation delay between two neurons for standard synapses. T_{neu} is the time needed by a neuron to emit a spike when triggered by an input synaptic event; it can model, for instance, timesteps of a neural simulator. In the experiments presented in this work, we use $T_{\text{syn}} = 1\text{ms}$ and $T_{\text{syn}} = 10\text{us}$. We define w_e as the minimum excitatory weight for $V-$synapses required to trigger a neuron in its reset state, and w_i the inhibitory weight of counteracting effect:

$$w_e = V_t \tag{6.7}$$
$$w_i = -w_e \tag{6.8}$$

Standard weights for g_e-synapses will be defined in the next subsection.

6.2.2 Signal representation

The main idea of the method proposed in this work is to represent values as the precise time interval in between two spikes.

If the series $e_n(i)$ is the list of times at which neuron n emitted spikes, with i the index of the spike in the series, neuron n encodes the signal $u(t)$ by:

$$u(e_n(i)) = f^{-1}(e_n(i+1) - e_n(i)), \forall i = 2.p,\ p \in \mathbb{N}, \tag{6.9}$$

with i an even number and f^{-1} the inverse of the encoding function f of our choice.

The encoding function $f : \mathbb{R} \to \mathbb{R}$ can be chosen depending on the considered signals in a particular system and adapted to the required precision. f computes the interspike time Δt associated

with a particular value. In the following work, we chose to represent values using the following linear encoding function:

$$\Delta t = f(x) = T_{\text{min}} + x.T_{\text{cod}}, \tag{6.10}$$

with $x \in [0,1]$ and T_{cod} the elementary time step.

This representation allows us to encode any value between a minimum and a maximum interspike (of T_{min} and $T_{\text{max}} = (T_{\text{min}} + T_{\text{cod}})$). We chose to use a minimum interspike to encode zero for several reasons. If the two spikes encoding a value originate from one unique neuron or are received by a single neuron, this minimum interspike gives time to recover from the first spike before spiking again. T_{min} allows networks to react to the first input spike and propagate a state change before the second encoding spike is received. In the experiments presented in this work, we use $T_{\text{min}} = 10\text{ms}$ and $T_{\text{cod}} = 100\text{ms}$.

We could also choose a logarithmic function to allow encoding a large range of values with dynamic precision (precision would be smaller for large values).

To represent signed values, we use two different pathways for the two different signs. Positive values will be encoded by causing a neuron to spike and negative values by eliciting another neuron to spike. We arbitrary chose to represent zero as a positive value.

To ease the understanding and the routing of several networks, each implementing simple operations, we add some interface neurons to these networks. For instance, the input of the networks are materialized by special *input* neurons. Their output by some *output*, *output* + or *output* - neurons. Other special neurons used for interaction between networks are also marked. In all the following figures describing networks, neurons in blue will be input neurons to the networks whereas neurons in red will be output neurons of the circuit.

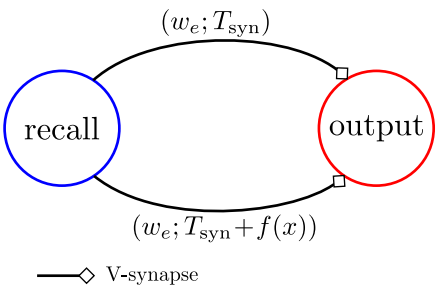


Figure 6.1: A network generating a constant value when required. When the *recall* neuron is activated, the *output* neuron will generate a pair of spikes coding for value x.

The simple network presented in Fig. 6.1 encodes a constant value. It shows the design principles which will be used in the rest of this section. In the network shown in Fig. 6.1, the *recall* neuron is an input. When a spike is received by *recall*, the constant value encoded in the network is output to the *output* neuron. In this example, the output is generated by two different synaptic connections from *recall* to *output*. They generate two output spikes with the interspike corresponding to the encoded value.

We define standard weights for $g_e - synapses$. Let w_{acc} be the weight value for g_e synapses to cause a neuron to spike from its reset state after time $T_{\text{max}} = T_{\text{min}}$+$T_{\text{cod}}$. According to Eq. 6.1, we have:

$$V_t = \frac{w_{\text{acc}}}{\tau_m}.T_{\text{max}} \tag{6.11}$$

such that:

$$w_{\text{acc}} = V_t \cdot \frac{\tau_m}{T_{\max}} \tag{6.12}$$

We also define weight $\bar{w}_{\text{acc}}$ as the g_e value necessary to have a neuron spiking from its reset state after time T_{cod}. The same equation gives us:

$$\bar{w}_{\text{acc}} = V_t \cdot \frac{\tau_m}{T_{\text{cod}}} \tag{6.13}$$

We can now describe different neural networks implementing elementary operations such as : memories, synchronizer, linear combination or non-linearities such as multiplications, directly operating on inter spike intervals.

6.2.3 Storing data: memory

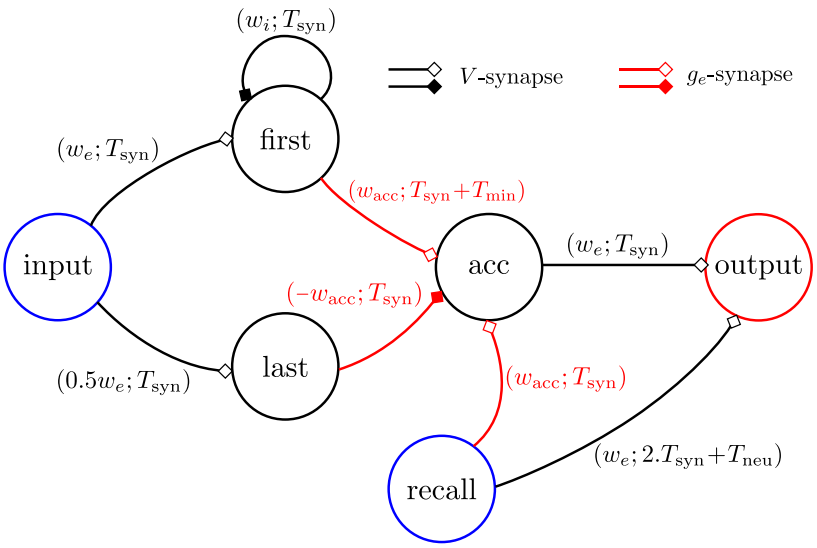


Figure 6.2: Inverting Memory: this network stores a value encoded in a pair of input spikes x (to the *input* neuron) by integrating current on the g_e dynamics of neuron *acc* during the input interspike. The value, stored in the membrane potential of the *acc* neuron is read out when the *recall* neuron is triggered. Releasing a pair of spikes with neuron *output* corresponding to $1-x$.
Blue, red and black neurons are, respectively, input, output and internal neurons.

Inverting Memory Fig. 6.2 presents an Inverting Memory network. This network is constituted of two input neurons (in blue): *input* and *recall*, one output neuron (in red): *output* and 3 internal neurons (in black): *first*, *last* and *acc*. Details and proof of the inner working of the network can be found in Appendix D.1.1. The chronogram of spikes during operation of this network is presented in Fig. 6.3. When a pair of spikes arrives at the *input* neuron, they are sorted by the *first* and *last* neurons. Their synaptic connections are such that *first* will only spike in response to the first encoding spike of the pair (at time t_{in}^1) and *last* will only spike in response to the second encoding spike of the pair (at time t_{in}^2), thus seperating the two input spikes.

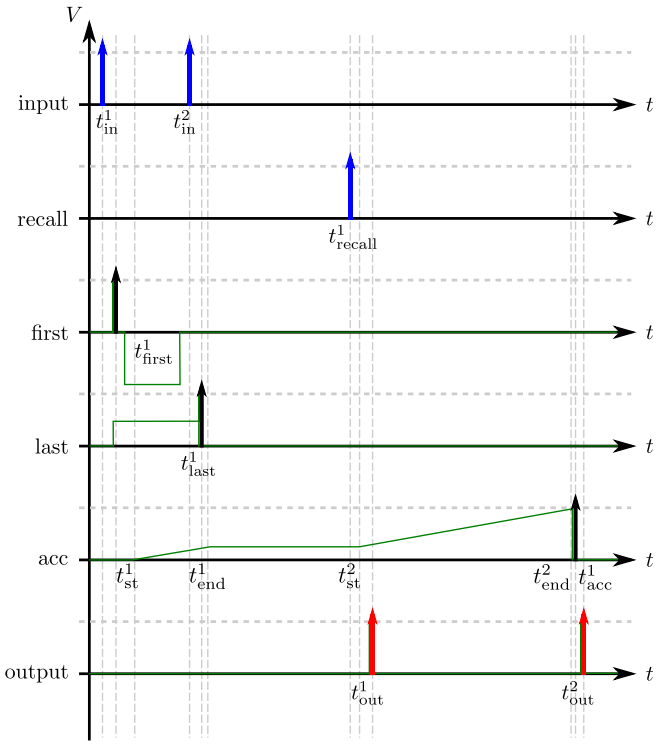


Figure 6.3: Inverting Memory: chronogram of network operation for an input at times t^1_{in} and t^2_{in} and a recall at time t^1_{recall}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neurons.)

first and *last* are then respectively starting and stopping the integration of the membrane potential of neuron *acc* at times t^1_{st} and t^1_{end} such that the value of *acc*'s membrane potential after the second input spike is, with $\Delta T_{in} = t^2_{\text{in}} - t^1_{\text{in}}$ the input interspike:

$$V_{sto} = \frac{w_{\text{acc}}}{\tau_m}.(\Delta T_{in} - T_{\text{min}}) \tag{6.14}$$

When the *recall* neuron is triggered, the integration starts again until reaching V_t such that we get an output interspike $\Delta T_{out} = t^2_{\text{out}} - t^1_{\text{out}}$ following:

$$V_t = \frac{w_{\text{acc}}}{\tau_m}.(\Delta T_{in} - T_{\text{min}}) + \frac{w_{\text{acc}}}{\tau_m}.\Delta T_{out} \tag{6.15}$$

Considering the definition of w_{acc}, we have $V_t.\tau_m/w_{\text{acc}} = T_{\text{max}}$, such that

$$\Delta T_{out} = T_{\text{max}} - (\Delta T_{in} - T_{\text{min}}) \tag{6.16}$$

We thus obtain an output spike corresponding to the maximum temporal representation of a value (T_{max}) minus the actual coding time ($\Delta T_{in} - T_{\text{min}}$) of the input value. Chaining two of these Inverting Memory networks, can store and recall a value without modification.

We can also notice that the value is stored in the inter spike timing needed to represent the value and can be recalled as soon as the value has been completely fed into the Inverting Memory network.

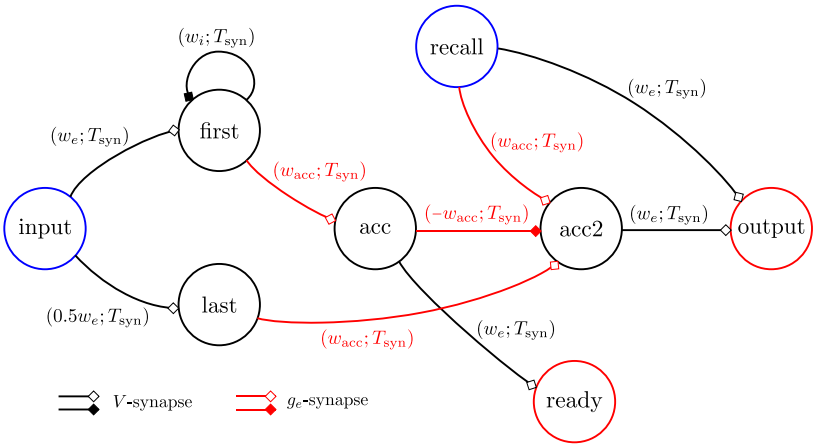


Figure 6.4: Memory: this network stores a value stored in the inter spike timing of two input spikes (to the *input* neuron) similarly to the Inverting Memory network. Two accumulators, *acc* and *acc2* are used to output on the *output* neuron the same value as previously received on the *input* neuron. Blue, red and black neurons are, respectively, input, output and internal neurons.

Memory Fig. 6.4 presents a Memory network. Detailed explanations and a chronogram of the network's operations can be found in Appendix D.1.2. This network is based on the Inverting Memory network introduced in the previous paragraph, 2 accumulator neurons *acc* and *acc2* are added to invert the stored value twice. If we follow the same reasoning as in the previous paragraph, *acc* spikes T_{max} after the first encoding spike is received and *acc2* starts integrating when the second encoding spike is received. Because *acc* stops *acc2*'s integration process, the value stored in *acc2*'s membrane potential after *acc* spikes is, with ΔT_{in} the input interspike (we present here a simplifyed result to ease the notations, the full result is available in Appendix D.1.2):

$$V_{sto} = \frac{w_{\text{acc}}}{\tau_m}.(T_{\text{max}} - \Delta T_{in}) \tag{6.17}$$

Note that the time at which *acc2* ends its integration happens after the input value has been completely fed into the Memory network. This is the reason why we added the *ready* neuron which is triggered when the input value has been stored in the Memory network and is ready to be read out. We then have the output interspike ΔT_{out} following:

$$V_t = \frac{w_{\text{acc}}}{\tau_m}.(T_{\text{max}} - \Delta T_{in}) + \frac{w_{\text{acc}}}{\tau_m}.\Delta T_{out} \tag{6.18}$$

such that

$$\Delta T_{out} = \Delta T_{in} \tag{6.19}$$

Signed Memory Fig. 6.5 presents a Signed Memory network. This network uses a Memory network to store a value and a small state machine, implemented by neurons *ready* + and *ready* -, to store the sign of the input. Detailed equations and the chronogram of the network's operations can be found in Appendix D.1.3. The internal Memory network is linked in parallel to the positive and negative pathways of the Signed Memory. When an input is fed into one of these two pathways, only the corresponding *ready* neuron receives some excitation. When the *recall* neuron

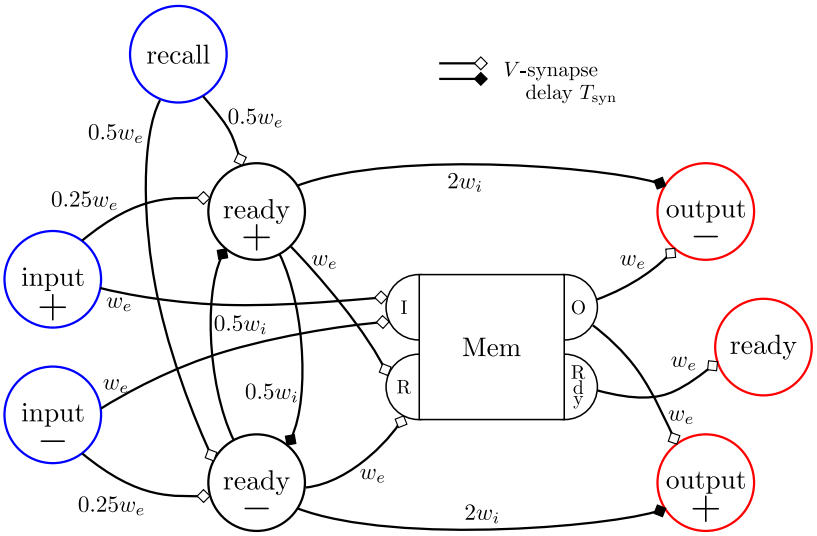


Figure 6.5: Signed Memory: this network is using a Memory network to store a signed value and its sign. Depending on the sign of the input (i.e. if it is received on the *input* + or *input* - neuron), the wrong output neuron (*output* + if the input is negative) is inhibited so that the output of the internal Memory network is directed to the output corresponding to the input sign. Blue, red and black neurons are, respectively, input, output and internal neurons.

is triggered, only the *ready* neuron corresponding to the sign of the input spikes (because of the excitation contributed by the stored input). The *ready* neuron will then inhibit the wrong output such that only the *output* neuron corresponding to the correct sign will fire.

Synchronizer Fig. 6.6 presents a Synchronizer network. This network receives N different inputs and synchronizes their first encoding spikes on the output end. Detailed equations and the chronogram of the network's operations can be found in Appendix D.1.4. It is implemented using N Memory networks. The *sync* neuron keeps track of the number of received inputs. When all the N inputs have been received, this neuron spikes, starting the readout process of the different memories at the same time, thus synchronizing all the outputs.

Furthermore, the same principle can be used with Signed Memory networks to obtain a Signed Synchronizer network.

6.2.4 Relational operations

Minimum Fig. 6.7 presents a Minimum network. This network implements the minimum operation on 2 inputs. It outputs the smallest of its 2 inputs as well as an indicator of which input was the smallest one. If the two inputs, *input1* and *input2*, are synchronized and because our encoding function is increasing with its input value, the minimum of the 2 inputs is the one for which the second encoding spikes arrives first. This is what the *smaller1* and *smaller2* neurons are extracting. This information is also used to inhibit the excitatory contribution of the largest input to the *output* neuron in order to output only the smallest of the input values. Detailed proof and the chronogram of operations can be found in Appendix D.2.1.

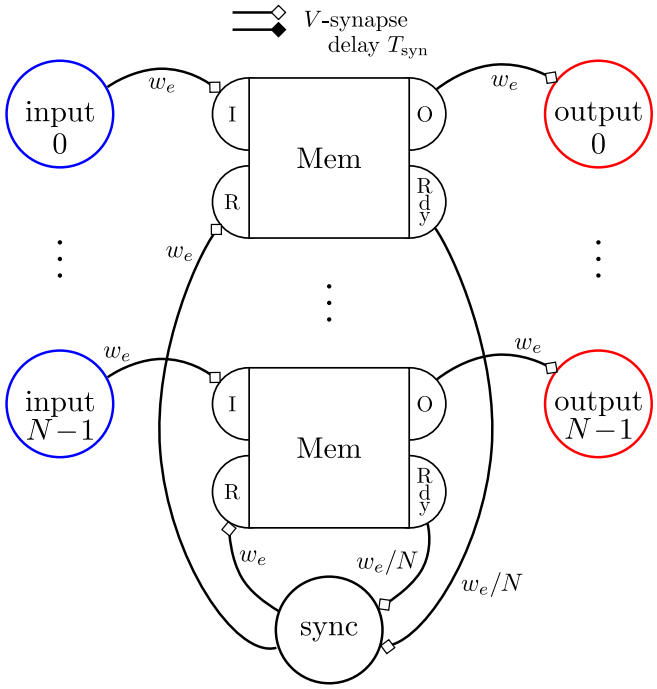


Figure 6.6: Synchronizer: this network is synchronizing a set of input values so that the first spikes encoding every value on its output side happen at the exact same time. It can be used to resynchronize values before networks requiring this condition. It uses a set of Memory networks to store the different input values and a *sync* neuron which recalls all these stored values as soon as the last one of them has been stored. Blue, red and black neurons are, respectively, input, output and internal neurons.

Maximum Fig. 6.8 implements the maximum operation on 2 inputs. This networks follows the same principle as in the Minimum network, it differs by inverting the detection relation: when the first input is the smallest, it triggers *larger2* because the second input must then be the largest. The drive of the *output* neuron is simpler in this case as the inputs are synchronized. The maximum value corresponds to the one for which the second encoding spike is the latest. Detailed proof and the chronogram of operations can be found in Appendix D.2.2.

6.2.5 Linear operations

Subtractor Fig. 6.9 presents a Subtractor network. It is here presented in its simplest form. It will be expanded in a second stage. This network computes the difference between *input1* and *input2* and directs the output depending on its sign either to the *output* + or *output* - neuron. If the two inputs are synchronized, the difference between the two is directly given by the interspike between the two second encoding spikes. This is the information *sync1* and *sync2* neurons are extracting. It also implements the same idea as in the Minimum network (see Fig. 6.7) to compute the sign of the output. When the output sign is known, *sync1* or *sync2* inhibits the pathway to the wrong output neuron such that the output spikes are directed to the correct one. The detailed proof and the chronogram of operations can be found in Appendix D.3.1.

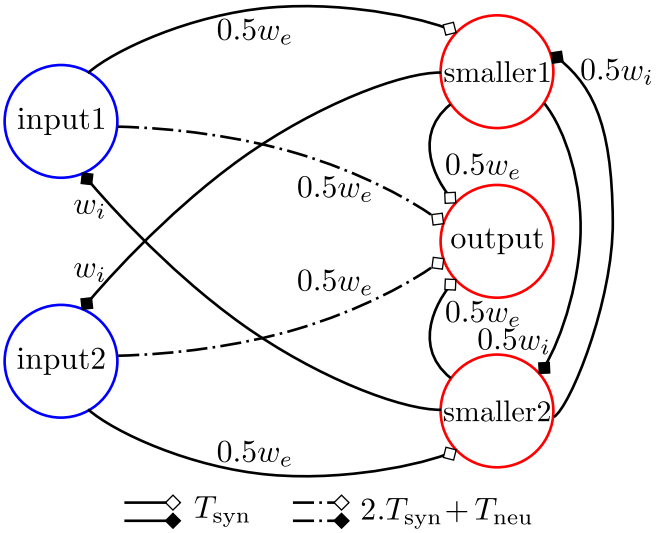


Figure 6.7: Minimum: this network outputs the smallest of its two inputs, which have to be synchronized, and an indicator signal on neuron *smaller1* or *smaller2* corresponding to which input has been considered as the smallest one. Because we chose an encoding function f which output increases with its input, the smallest input corresponds to the one for which the second encoding spike arrives first. This is what the *output* neuron is extracting. Blue, red and black neurons are, respectively, input, output and internal neurons. This network only contains $V-$ synapses

A more robust version of the Subtractor network is presented in Fig. 6.10. The networks adds the *zero* neuron and its connections (in magenta in the figure). In the previous 'simple' implementation shown in Fig. 6.9, when both inputs are equals, the two parallel pathways of the network are triggered at the same time and the lateral inhibition has no time to select a winning pathway. In that case, the output is emitted both on *output* + and *output* - which can be problematic for the following networks expecting only one of the two pathways to be activated. To solve this problem, we add the *zero* neuron with a set of fast synaptic connections. They allow to detect the case of equality between the two inputs. In this case, the *output* - pathway is quickly inhibited to produce spikes coding for the zero output only on the *output* + neuron.

Linear Combination Fig. 6.11 presents a Linear Combination network. It computes the linear combination of N signed inputs with arbitrary coefficients $\alpha_0, ...\alpha_{N-1}$:

$$s = \sum_{i=0}^{N-1} \alpha_i.x_i, \tag{6.20}$$

where $x_i, i \in 0, ..., N-1$ are the different inputs of the network. It uses the same principle as in the Memory network to store values in an accumulator. To implement the coefficients of the sum, we multiply the synaptic weight of the accumulation current by the coefficient corresponding to the input. To handle the signs of the inputs and coefficients, we use 2 accumulators. The first one is storing intermediate results which are positives (i.e. when the sign of the input is the same as the one of its associated coefficient) while the second stores negative values (i.e. when the sign

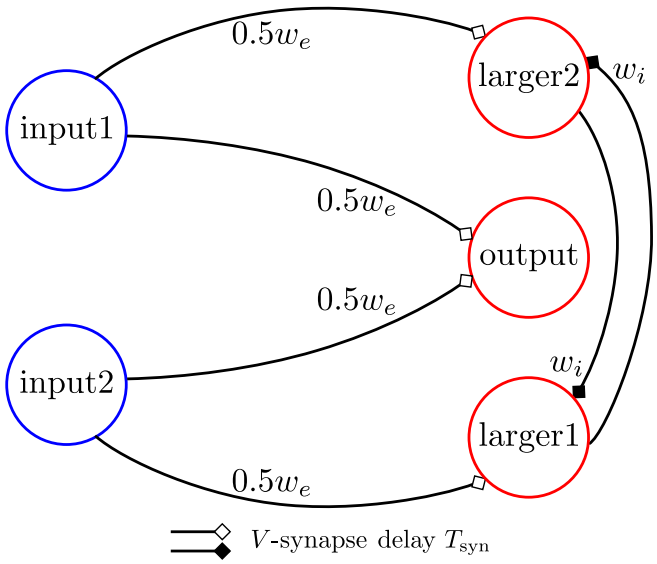


Figure 6.8: Maximum: this network outputs the largest of its two inputs, which have to be synchronized, and an indicator signal on neuron *larger1* or *larger2* corresponding to which input has been considered as the largest one. Because we chose an encoding function f which output increases with its input, the largest input corresponds to the one for which the second encoding spike arrives last. This is what the *output* neuron is extracting. Blue, red and black neurons are, respectively, input, output and internal neurons.

of the input is different from the one of its associated coefficient). When all the inputs have been fed into the network, the *sync* network is triggered, causing the readout process of these accumulators. Their content are inverted as for the Memory network and then synced before entering a Subtractor network. This last network computes the difference between the positive and negative contributions of the different inputs and produces a signed output. A *start* neuron is then triggered to spike to indicate that the computation has ended. This signal can be used to trigger further networks. All details and proofs can be found in Appendix D.3.2.

6.2.6 Non-linear operations

Natural Logarithm Fig. 6.12 presents a Log network capable of computing the natural logarithm of its input value by exploiting the dynamics of a $g_f -$ synapse. Detailed proof and the chronogram of operations can be found in Appendix D.4.1. When an input value is fed into the *input* neuron, this value is first stored in the membrane potential of *acc* by integrating with weight $\bar{w}_{\text{acc}}$ between the spike of *first* and *last*. Because the spike from *first* is delayed by an additional $T_{\min}$ compared to the one from *last*, the value stored in *acc*'s membrane potential is, with $\Delta T_{in} = T_{\min} + \Delta T_{cod}$ the input interspike:

$$V = \frac{\bar{w}_{\text{acc}}}{\tau_m}(\Delta T_{in} - T_{\min}) = \frac{\bar{w}_{\text{acc}}}{\tau_m}.\Delta T_{cod} = V_t.\frac{\Delta T_{cod}}{T_{\text{cod}}} \tag{6.21}$$

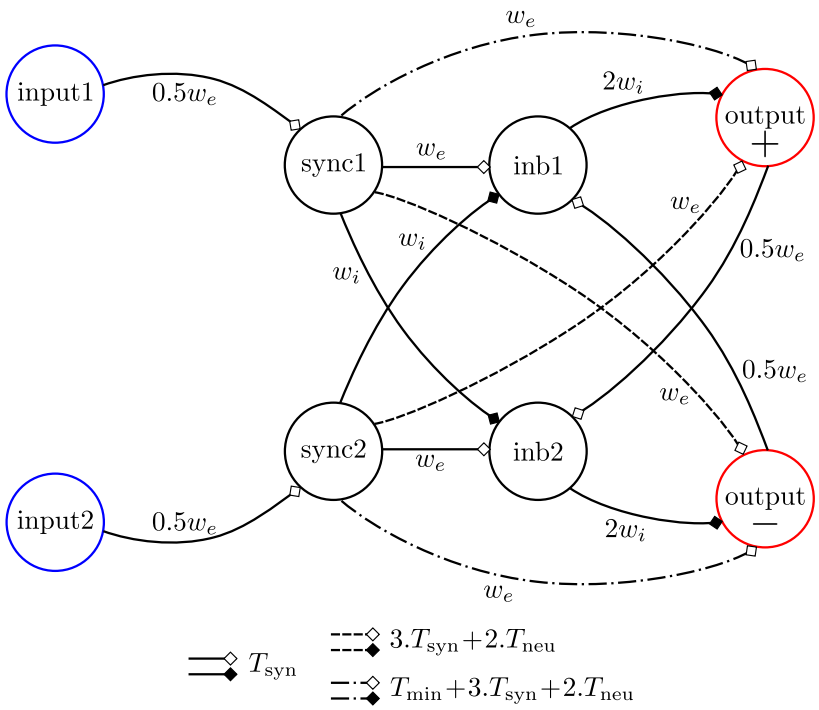


Figure 6.9: Subtractor (simple): this network substracts its two inputs, which have to be synchronized. The resulting value is directed to two different neurons, *output* + or *output* -, depending on its sign. Because the two inputs are synchronized, the difference is directly given by the inter-spike between the second spikes of both inputs. The sign is determined using the same idea as in the Minimum network (see Fig. 6.7). Blue, red and black neurons are, respectively, input, output and internal neurons. This network only contains $V-$ synapses

When this integration process is stopped by *last*'s spike, a synaptic event is also triggered on a g_f- synapse of *acc* with weight g_{mult} (another synaptic event also enables the g_f dynamics by activating the *gate* state). *acc*'s membrane potential thus follows the evolution given by solving the differential system Eq. (6.1):

$$V = V_t.\frac{\Delta T_{cod}}{T_{\text{cod}}} + g_{\text{mult}}\frac{\tau_f}{\tau_m}(1-e^{-t/\tau_f}) \tag{6.22}$$

If we chose g_{mult} such that:

$$g_{\text{mult}} = V_t.\frac{\tau_m}{\tau_f} \tag{6.23}$$

We obtain

$$V = V_t.\frac{\Delta T_{cod}}{T_{\text{cod}}} + V_t.(1-e^{-t/\tau_f}) \tag{6.24}$$

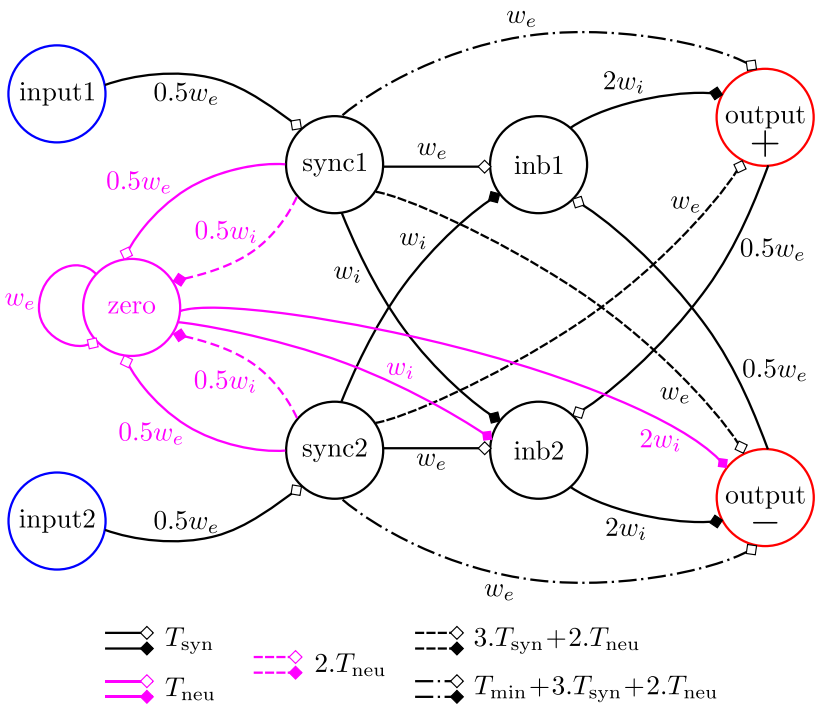


Figure 6.10: Subtractor (full): this nework implements the same operation as the one presented in Fig. 6.9. The simple version requires that the input are not equal so that the inhibition resulting from the sign determination has time to propagate. The added *zero* neuron and its connections (in magenta) detects this particular case and drives the network to output a zero on neuron *output* + if the two inputs are equal. Blue, red and black neurons are, respectively, input, output and internal neurons. This network only contains $V-$ synapses

Considering neuron *acc* will then spike at time t_s where $V = V_t$, we get:

$$V_t = V_t.\frac{\Delta T_{cod}}{T_{\text{cod}}} + V_t.(1-e^{-t_s/\tau_f}) \tag{6.25}$$

$$\frac{\Delta T_{cod}}{T_{\text{cod}}} = e^{-t_s/\tau_f} \tag{6.26}$$

$$t_s = -\tau_f.\log\left(\frac{\Delta T_{cod}}{T_{\text{cod}}}\right) \tag{6.27}$$

which is a positive value because $\Delta T_{cod} \leq T_{\text{cod}}$ by definition. Adding the delays of the synaptic connections to the *output* neuron, we get an output interspike ΔT_{out}:

$$\Delta T_{out} = T_{\text{min}} + \tau_f.\log\left(\frac{T_{\text{cod}}}{\Delta T_{cod}}\right) \tag{6.28}$$

We thus obtain a network capable of generating an output proportional to the natural logarithm of its input.

Exponential Fig. 6.13 presents the Exp network that computes the exponential of its input value by exploiting the dynamics of a g_f- synapse. The detailed proof and the chronogram of operations

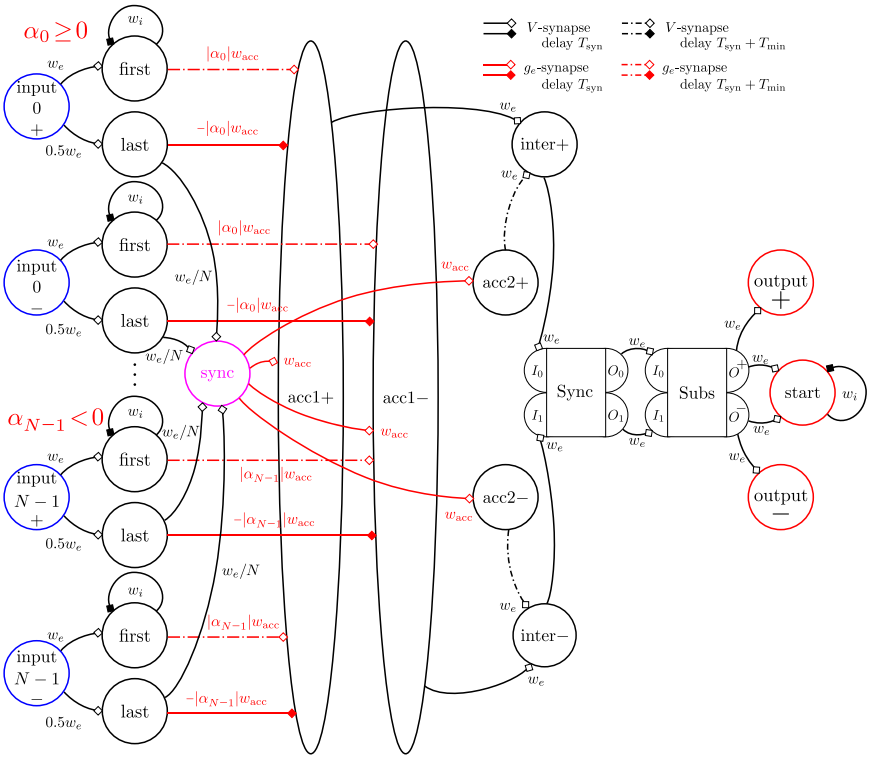


Figure 6.11: Linear Combination: this network computes the linear combination of a set of inputs with given coefficients $\alpha_0, ..., \alpha_{N-1}$: $s = \sum_{i=0}^{N-1} \alpha_i.x_i$. The network is accumulating positive and negative sub-operations (i.e. $\alpha_i.x_i$) in 2 distinct accumulators. Synaptic connections in the network are shown for $\alpha_0 \geq 0$ and $\alpha_N < 0$. The two resulting values are then read out by the *sync* neuron when all inputs are known, synchronized and substracted from one another to obtain the full, signed, result s. An indicator neuron *start* is provided to notify when the result is ready. Blue, red and black neurons are, respectively, input, output and internal neurons.

can be found in Appendix D.4.2. When an input value is fed into the *input* neuron, a synaptic event is triggered on a g_f − synapse of the *acc* neuron with weight g_{mult} as defined in Eq. 6.23 by neuron *first*. *acc*'s membrane potential thus follows the evolution given by solving the differential system Eq. 6.1 until *last* spikes:

$$V = g_{\text{mult}} \frac{\tau_f}{\tau_m} (1 - e^{-t/\tau_f}) = V_t.(1 - e^{-t/\tau_f}) \tag{6.29}$$

This synaptic activity is then gated when neuron *last* is triggered and blocks *acc*'s membrane potential value to :

$$V = V_t.(1 - e^{-\Delta T_{cod}/\tau_f}), \tag{6.30}$$

with $\Delta T_{in} = T_{\text{min}} + \Delta T_{cod}$ the input interspike (Because of the additional delay of T_{min} in *first*'s pathway in comparison to the one of *last*). The spiking of *last* also triggers a readout of *acc*'s membrane potential by initiating a g_e synaptic event with weight $\bar{w}_{\text{acc}}$ such that *acc* spikes after

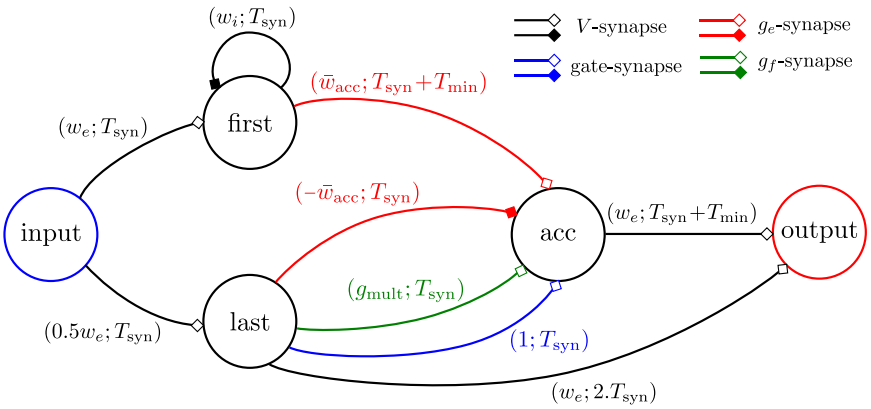


Figure 6.12: Log: this network computes the natural logarithm of its input by using the g_f dynamics of neuron *acc*. The input value is first stored in the membrane potential of the *acc* neuron. When the second encoding spike arrives, a $g_f -$ synapse is used to obtain a delay function of the log of the current value of *acc*'s membrane potential. Blue, red and black neurons are, respectively, input, output and internal neurons.

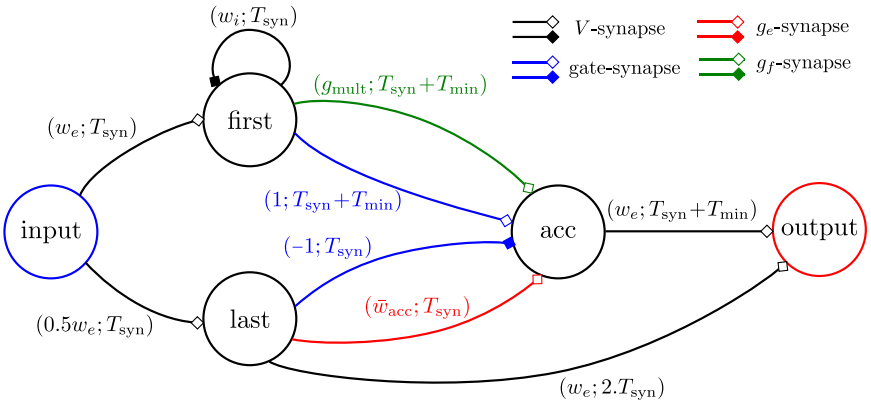


Figure 6.13: Exp: this network computes the exponential of its input by using the g_f dynamics of neuron *acc*. A value function of the exponential of the input value is first stored in neuron *acc*'s membrane potential by gating a $g_f -$ synapse to *acc* on the second encoding spike of the input. *acc*'s membrane potential is then read out to obtain the result. Blue, red and black neurons are, respectively, input, output and internal neurons.

time t_s following:

$$V_t = V_t.(1-e^{-\Delta T_{cod}/\tau_f}) + \frac{\bar{w}_{\text{acc}}}{\tau_m}.t_s \tag{6.31}$$

$$V_t = V_t.(1-e^{-\Delta T_{cod}/\tau_f}) + V_t.\frac{t_s}{T_{\text{cod}}} \tag{6.32}$$

$$t_s = T_{\text{cod}}.e^{-\Delta T_{cod}/\tau_f} \tag{6.33}$$

Adding the delays of the synaptic connections to the *output* neuron, we get an output interspike ΔT_{out}:

$$\Delta T_{out} = T_{\min} + T_{\text{cod}}.e^{-\Delta T_{cod}/\tau_f} \tag{6.34}$$

We thus obtain a network capable of generating an output proportional to the exponential of its input.

Going back to the Log network from the previous paragraph, its output was :

$$\Delta T_{cod}^{\log} = \Delta T_{out}^{\log} - T_{\min} = \tau_f.\log\left(\frac{T_{\text{cod}}}{\Delta T_{cod}}\right) \tag{6.35}$$

If this output is fed into an Exp network, we obtain the output:

$$\begin{aligned} \Delta T_{out}^{\exp} &= T_{\min} + T_{\text{cod}}.e^{-\Delta T_{\text{cod}}^{\log}/\tau_f} && (6.36)\\ &= T_{\min} + T_{\text{cod}}.e^{\log(\Delta T_{cod}/T_{\text{cod}})} && (6.37)\\ &= T_{\min} + \Delta T_{cod} && (6.38) \end{aligned}$$

The Exp network is thus capable of inverting the Log network. One can take advantage of that to implement several common non-linearities by applying simple operations in between the Log and Exp networks. For instance, summing two logarithms will allow multiplication, subtracting them will implement division, multiplying a logarithm by a constant will compute a power function, ...

Multiplier Fig. 6.14 presents a network we name Multiplier. The detailed proof and the chronogram of the network's operations can be found in Appendix D.1.3. This network is based on the principles used in the Log and Exp networks. The product s of the 2 inputs x_1 and x_2 is obtained using the well known following equation:

$$s = x_1.x_2 = \exp(\log x_1 + \log x_2). \tag{6.39}$$

Neurons *acc_log1* and *acc_log2*'s membrane potentials are first loaded with the two inputs (*input1* and *input2*). When the 2 inputs have been received, an exponential circuit is triggered through *acc_exp*. To obtain the product of the input, this circuit has to be stopped after a time corresponding to the sum of the natural logarithm of the 2 inputs. Because the absolute value of the natural logarithm can be larger than 1 for small inputs (i.e. it is larger than the maximum value representable by our encoding scheme), we cannot use a Linear Combination network to sum the logs. To overcome this problem, we sum these values by triggering the logarithm computation of the 2 inputs successively: the *sync* neuron, which is detecting the end of the second input, it activates at the same time the exponential circuit and the logarithm of the first input. When the first input's logarithm is output, it triggers the logarithm of the second input which, when computed, stops the exponential circuit. At that point in time, the *acc_exp* neuron contains in its membrane the product of the 2 inputs. It is then read out to compute the actual output of the network.

Signed Multiplier Fig. 6.15 presents a Signed Multiplier network. It computes the product of two inputs independently of their sign. In parallel, a set of neurons: *sign1*, *sign2*, *sign3* and *sign4* are used as a truth table to determine the sign of the output from the sign of the inputs. When

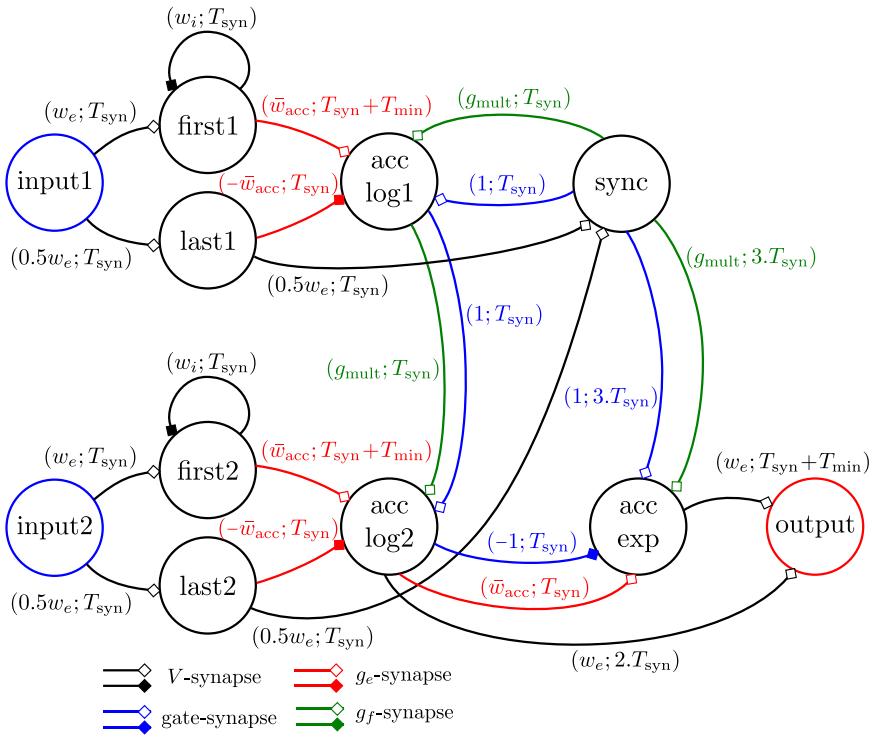


Figure 6.14: Multiplier: this network multiplies its two inputs. It computes the logs of its inputs through neurons *acc_log1* and *acc_log2*, sum them up and gets the exp of this value through neurons *acc_exp* to obtain the product : $s = x_1.x_2 = \exp(\log x_1 + \log x_2)$. Blue, red and black neurons are, respectively, input, output and internal neurons.

the output sign is known, the wrong output pathways (either *output* + or *output* -) is inhibited to direct the output of the Multiplier network to the right output neuron. The different *sign* neurons implement a truth table with the excitatory connections they receive from the input neurons. Input connections and weights are designed such that only one *sign* neurons spikes when an input is fed into the circuit. This "winning" neuron can then be associated to an output sign. Lateral inhibition between the *sign* neurons is present to suppress residual activations by the input of non-winning *sign* neurons (this allows all the *sign* neurons to go back to their reset state once the output sign is computed).

6.2.7 Differential equations

Integrator Fig. 6.16 presents a Integrator network that allows to reconstruct a signal from its derivative fed into its input. It is using a multiplier and an accumulation network with a Linear Combination network. The output of this accumulator network is looped into its first input with a unit gain. The input, composed of *input* + and *input* -, is fed into this accumulator with a gain *dt* corresponding to the chosen integration timestep. Each time an output is produced on *output* + and *output* -, the indicator neuron *new_input* is triggered to notify that the integrator is ready to receive its next input. This system is thus driven by its input: every time an input is provided, the corresponding output is computed. Two auxiliary input neurons are also provided. The *init* neuron

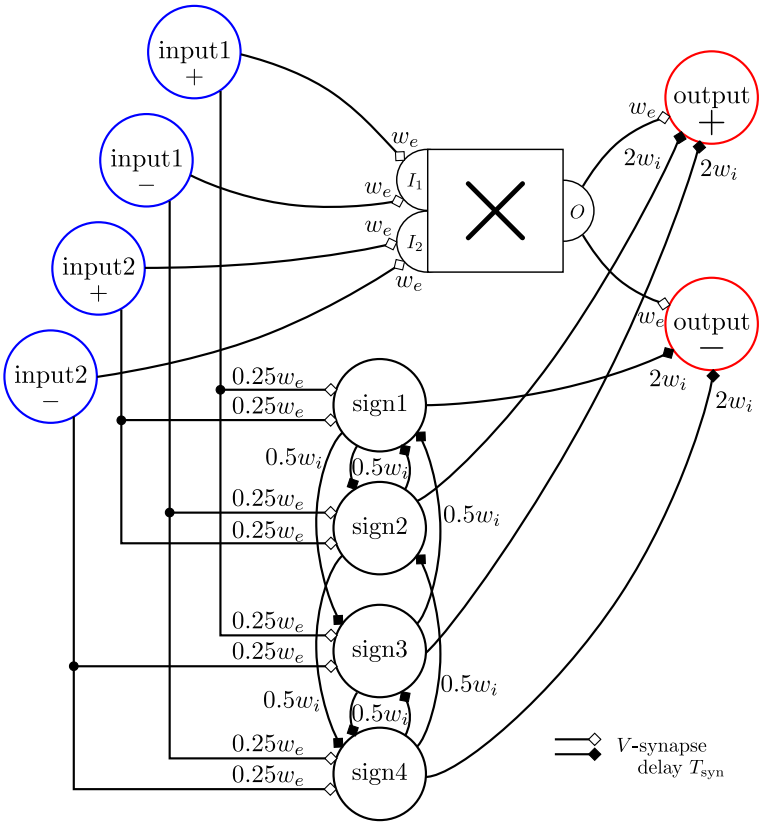


Figure 6.15: Signed Multiplier: this network computes the result and the sign of the multiplication of its two inputs. A Multiplier network is used to compute the absolute value of the result. This value is then directed to the correct output (*output* + or *output* -) by a small truth-table implemented in neurons *sign1*, *sign2*, *sign3* and *sign4* which are determining the sign of the output from the signs of the two inputs. Blue, red and black neurons are, respectively, input, output and internal neurons.

loads the integrator with its initial value. This allows the internal state of the integrator (through the Linear Combination network) to be set to an initialization value. The *start* neuron feeds a zero into the input of the integrator, thus computing its first output.

System design All the networks presented in this section can be assembled to achieve more complex computational tasks. Multiplier and Linear Combination networks can be associated to compute arbitrary functions on some state variables. Integrator networks can then be used to solve systems of differential equations. Examples of such network will be demonstrated in the next section.

6.3 Results

We implement in this section different computational tasks. We start by implementing linear differential equations with a first order and a second order system. In a second stage, we implement

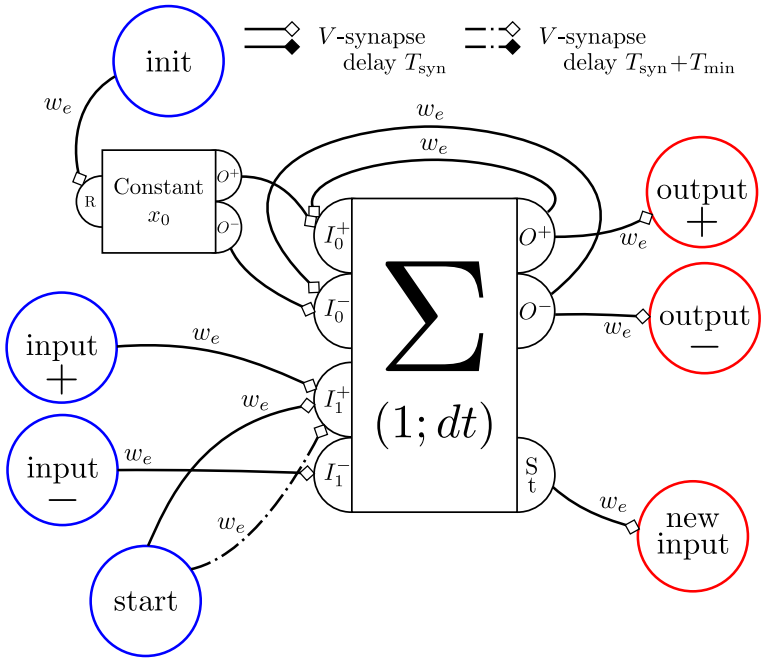


Figure 6.16: Integrator: this network integrates an input over time. The *init* and *start* neurons are used to initialize the inner state of the integrator. The network is then driven by its input. Everytime an input is received, it is integrated and added to the latest value of the output. When a new value is output, the *new_input* neuron spikes, requesting for a new input. Blue, red and black neurons are, respectively, input, output and internal neurons.

a more complex set of non-linear differential equations from Edward Lorenz.

6.3.1 Linear differential equations

First order system We first implement a first order differential system. This system solves the following equation:

$$\tau.\frac{dX}{dt} + X(t) = X_\infty \tag{6.40}$$

We implement this network as shown in Fig. 6.17 using 3 of the networks described in the previous section:

- a Constant network is providing the input X_∞ to the system,
- a Linear Combination network is computing dX/dt,
- an Integrator network is computing X from its derivative.

The *init* and *start* neurons enable to initialize and start the integration process. *init* has to be triggered before the integration process can take place to load the initial value of the Integrator network. *start* has to be triggered to output the first value from the Integrator network. When an output is provided by the Integrator network, the Constant network is activated using the *new_input* neuron of the Integrator. Hence feeding two values into the Linear Combination computing the

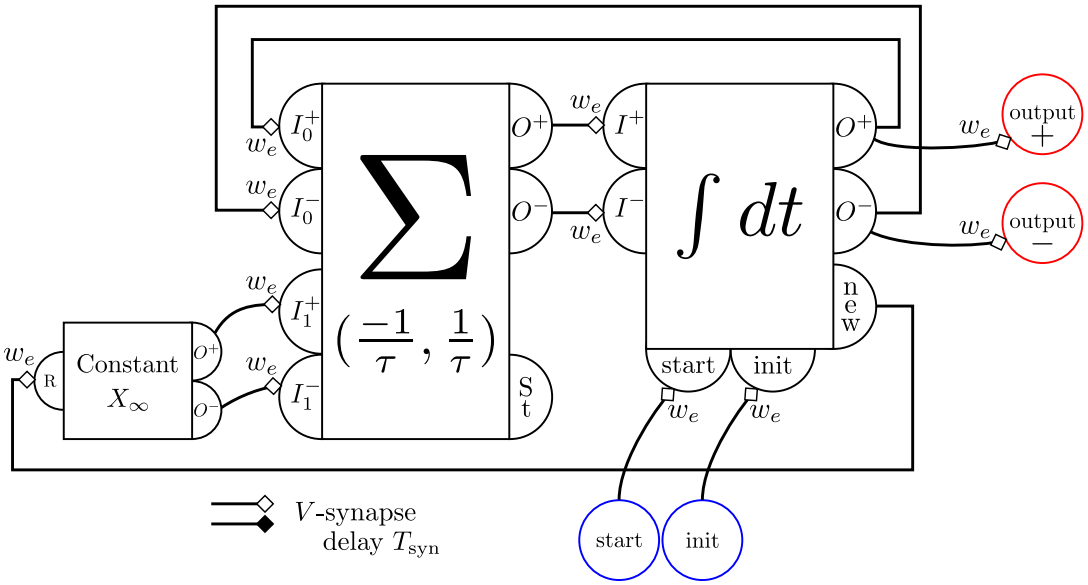
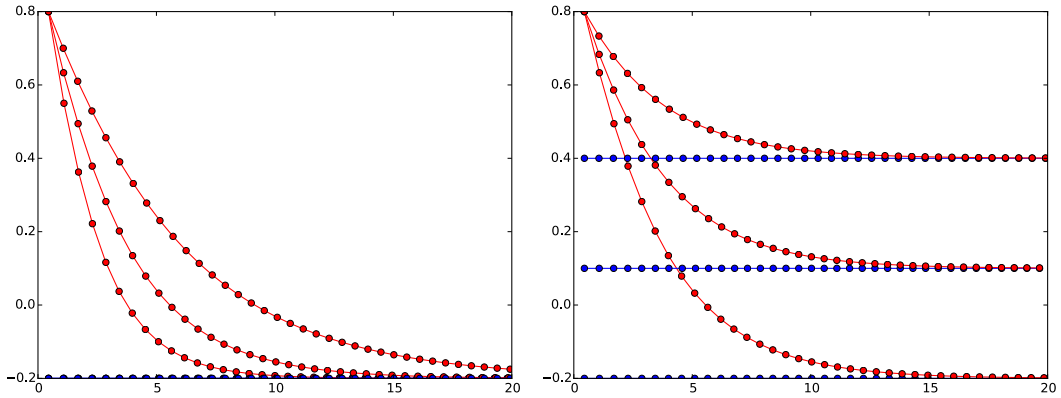


Figure 6.17: Neural network implementing a first order differential equation. It is composed of a Constant network providing the input, a Linear Combination network computing the derivative of X and an Integrator network computing X from $\mathrm{d}X/dt$.



(a) Different values of τ (b) Different values of X_∞

Figure 6.18: Neural implementation of a first order filter. Blue curves show the input values to the filter, red curves show the output values of the filter for different parameters. (Horizontal axis is time, every circle correspond to an actual output encoded by spikes of the network.)

new derivative of the output. This derivative is then integrated by the Integrator to obtain a new output. With the implementations presented in the previous section, this network requires 118 neurons. Results of its simulation with different set of parameters for τ and X_∞ and for $dt = 0.5$ are presented Fig. 6.18.

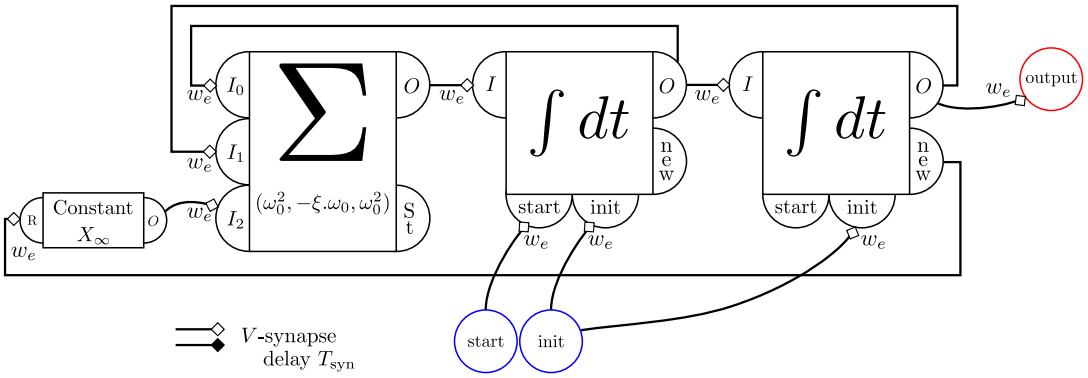
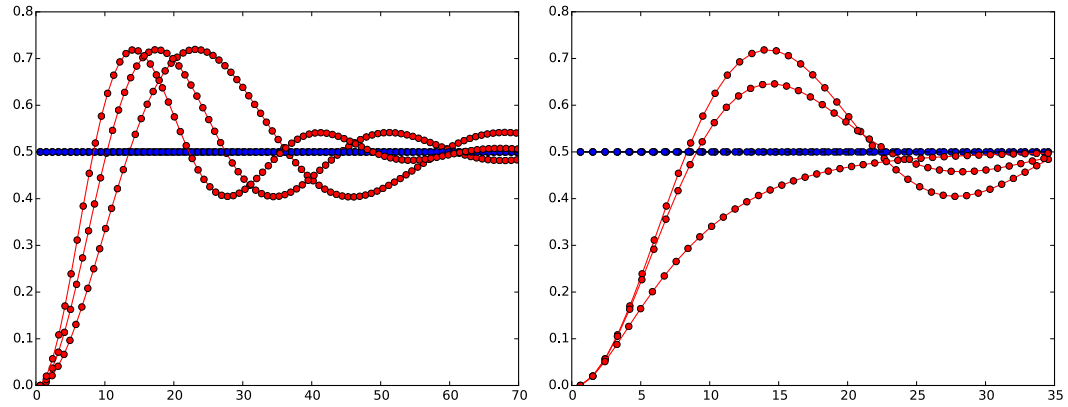


Figure 6.19: Neural network implementing a second order differential equation. It is composed of a Constant network providing the input, a Linear Combination network computing the second derivative of X and two Integrator networks computing $\mathrm{d}X/dt$ and X from d^2X/dt^2. (Signs of the different signals have been omitted to increase readability.)



(a) Different values of ω_0

(b) Different values of ξ

Figure 6.20: Neural implementation of a second order filter. Blue curves show the input values to the filter, red curves show the output values of the filter for different parameters. (Horizontal axis is time, every circle correspond to an actual output encoded by spikes of the network.)

Second order system To add complexity, we now implement a second order differential system. This system solves the following equation:

$$\frac{1}{\omega_0^2}.\frac{\mathrm{d}^2X}{dt^2} + \frac{\xi}{\omega_0}.\frac{\mathrm{d}X}{dt} + X(t) = X_\infty \tag{6.41}$$

We implement this network as shown in Fig. 6.19 using 4 of the networks described in the previous section:

- a Constant network is providing the input X_∞ to the system,

- a Linear Combination network is computing $\frac{d^2X}{dt^2}$,
- a first Integrator network is computing dX/dt from the second derivative of X,
- a second Integrator network is computing X from its first derivative.

With the implementations presented in the previous section, this network requires 187 neurons. Results of its simulation with different set of parameters for ξ and ω_0 and for $dt = 0.2$ are presented Fig. 6.20.

6.3.2 Lorenz attractor

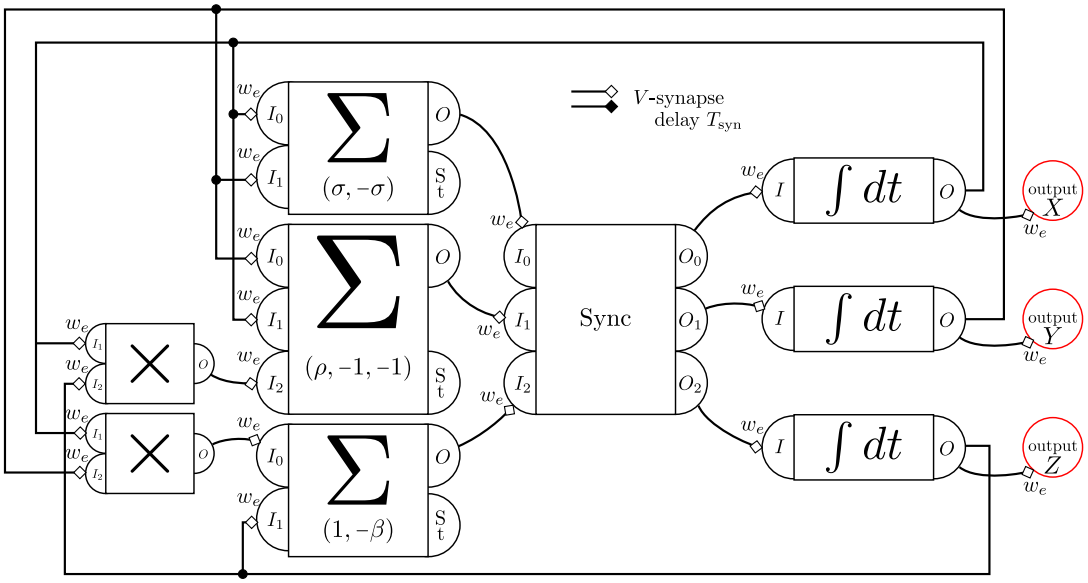


Figure 6.21: Neural network implementing Edward Lorenz's non-linear differential equation system. (Signs of the different signals have been omitted to increase readability.)

We now implement the set of non-linear differential equation proposed by [260]:

$$\frac{dX}{dt} = \sigma(Y(t) - X(t)) \tag{6.42}$$

$$\frac{dY}{dt} = \rho X(t) - Y(t) - X(t).Z(t) \tag{6.43}$$

$$\frac{dZ}{dt} = X(t).Y(t) - \beta Z(t) \tag{6.44}$$

using $\sigma = 10$, $\beta = 8/3$ and $\rho = 28$ to ensure chaotic behavior of the system. We also use a variable substitution to obtain state variables X, Y and Z evolving in $[0,1]$ so that they can be represented by our framework. The initial state of the system is set to $X = -0.15$, $Y = -0.20$ and $Z = 0.20$. We use an integral step of $dt = 0.01$.

We implement this network as shown Fig. 6.21 by using 9 of the networks described in the previous section:

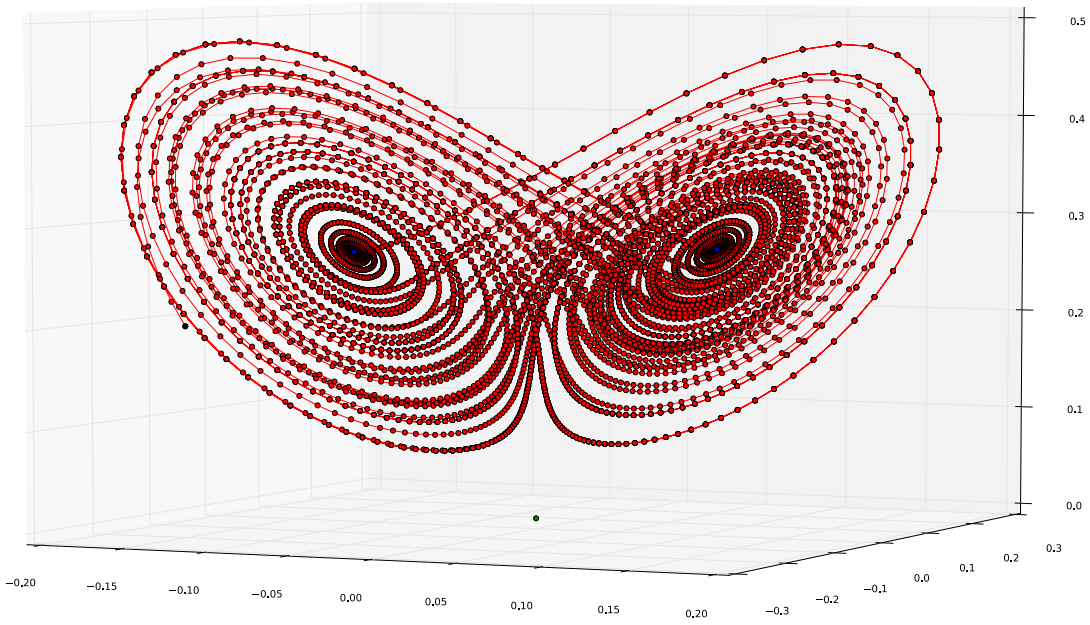


Figure 6.22: Neural implementation of a set of non-linear differential equations. These equations converge to the Lorenz attractor. The plot shows the evolution of the system in the phase space, every axis being one of the state variables X, Y, Z. Fixed points of the system are shown in blue, the origin point is shown in green. Every circle corresponds to an actual output encoded by spikes of the network.

- 2 Signed Multiplier networks compute the non-linarities contained in Y and Z's derivatives,
- 3 Linear Combination networks compute the derivatives of X, Y and Z,
- a Signed Synchronizer network allows to wait for the three derivatives to be computed before evolving the system's state,
- 3 Integrator networks compute the new state from the derivatives of X, Y and Z.

With the implementations presented in the previous section, this network requires 549 neurons. Results of its simulation are presented Fig. 6.22. We can observe that the system is behaving as expected, following the strange attractor described by Lorenz.

6.4 Discussion

In this work, we choose a linear encoding function to map values into inter spikes. In this case, it results in a direct trade-off between the time needed to represent a value and the time precision of the system. A finer time precision leads to a larger number of possible different values in a given maximum representation time. In this work we chose to set time scales that are compatible with neuroscience evidence. However, current hardware allows much faster time scales up to nanoseconds. In that case not only can we obtain higher precision but also faster computation times.

The number of neurons used in the current implementation of the examples given in this chaptercan be reduced in size. They have been designed to ease comprehension and to be easy to plug into each other. In almost all the networks, the first and second encoding spikes for the output are generated by different neurons of the network. All the networks have a pair of *first* and *last* neurons which task is to separate the two incoming spikes. By directly routing these two spikes independently, the whole implementation would be much more efficient and would require much less neurons. Less neurons would also imply less spikes and therefore less energy requirements and less latency in signal propagation.

For feedforward architectures, the different layers of computation could also be pipelined. This means that if a task is composed of a series of operations which can be considered as layers, the full operation would require data to go through all the layers, with all layers being active only a fraction of the time. But at every point in time, one layer could also be computing its operation on a different input, such that all of the layers are always active. In the end, one can get one output per pipeline period, even if the whole computation takes several pipeline period to be completed (as many pipeline period as there are steps in the computation, the pipeline period being the longest time needed by one stage to output its result). This means better throughput for feedforward computation, but imposes a minimum delay on feedback (because the result to be fed back is not available before a certain number of pipeline periods).

The developed framework can be used to compute any algorithm, it is intersting to notice that it is naturally compatible with all type of time oriented AER (Address Event Representation [30]) data and all kind of AER sensors [10]. The use of interspike makes the framework particularly adapted to process luminance time encoded events data from the neuromorphic camera ATIS ("Asynchronous Time-based Image Sensor") [13, 12]. Thus every event-based machine vision algorithm developed so far could be systematically implemented on neural boards.

A question outside the scope of this chapter relates to the architecture of the platform that should implement this computation paradigm. Several solutions could be possible. A pure analog chip could be used. Analog design is known to be a difficult endeavour, moreover we would need to robustify the framework to overcome mismatch. Several options are possible, the most straightforward is to use regularization techniques such as the one introduced in [19] or simply using calibration techniques to provide a precisely timed output from analog chip similar to what has been introduced in [261, 262].
A mixed-signal integrated circuit mixing both analog circuits and digital circuits could also be considered. Exponential decays could be generated using analog circuits while the remaining operations could use digital circuits. Finally the last solution could be a pure digital architecture preserving the principles introduced in this work such as inter spike encoding, local computations that allow to overcome the Von Neumann bottelneck. In this case, we could use local binary computation units to perform conventional computation rather than using membrane potentials, synapses, delay,.... Everything being local to the units, the architecture would still be efficient. This by far is the less elegant solution, but it could be a rapid and an intermediate step toward a full true neuromorphic implementation.

6.5 Conclusion

This work introduced a new clockless framework to build a multipurpose neuromorphic computer. Instead of representing values as a set of bits in a register or a part of some central memory, values are coded in the precise timing between events happening in the system. This dataflow framework we called STICK (Spike Time Interval Computational Kernel) offers a new method to design computing platforms where memory and computation are intertwined. By removing the numerous accesses to a central memory, inherent to standard computers, we free ourselves from the *Von Neumann* bottleneck. The systems scales naturally, the more neurons are available the more computation can be performed. Building a large machine consisting of several small elementary computational units allows to build natively massively parallel machines which rely on neuromorphic engineering principles to reduce their power consumption: energy is needed to produce events which only happens when information is produced.

Moreover, the STICK framework offers a way to design large neural networks that accomplish complex tasks by design of their architecture instead of only exploiting the randomness of a given connectivity topology. It also defines all the weights necessary to obtain a functional solution without the need for a costly (in terms of energy, time or computational power) learning process. A hardware fabric consisting of a large number of the proposed computational units with dynamic connection could then be used and adapted to successively solve different tasks.

Discussion & Conclusion

In all the work presented throughout this document, we showed that event-based vision has real potential. Even if standard computer vision has been obtaining interesting results for a long time now, we showed that a paradigm shift from images to events is offering many advantages.

Standard cameras enable computer vision to solve a lot of tasks, but this very quickly fails in uncontrolled conditions. The independance to luminance variation and the high dynamic range of asynchronous neuromorphic cameras allow to solve several problems faced by conventional machine vision. Moreover, the event-based acquisition process of such sensors, by adapting the sampling rate of data to their actual dynamics is optimizing the acquisition process. If some part of the visual field changes very slowly, it will be sampled at a low rate. If other pixels are observing a fast changing phenomenon, they will sample light at a much higher rate. This process allows to acquire a visual scene without under-sampling and over-sampling it at the same time.
The sparseness of the resulting data is a key component of the success of event-based algorithms. By only processing relevant information, it becomes possible to run algorithms at several kHz.

During my Ph.D. work, we have demonstrated that event-based vision is opening new perspectives in machine vision. The data representation allows for simpler algorithms which obtain better performances than the state of the art of computer vision. Because these algorithms are simpler, they can be computed much faster and consume less resources.

The next challenge is now to move from controlled scenes in the lab to unconstrained environments and real-world applications. Nowadays, the progress in event-based vision and processing are sufficient to initiate a movement towards the industry. Thinking without frames is new for most people but event-based vision is starting to be slowly accepted.

This is why we developed the kAER framework for event-based computing. This software framework is designed for use with standard computers[1]. It offers a standard way to implement and reuse event-based algorithms for realtime computation. Algorithms can be picked from a library of filters and assembled together to solve a particular task. Many different algorithms are available such as object tracking, motion flow, 3D reconstruction, ...

Filters can be automatically parallelized in a pipelined architecture to make use of the several processors of the machine it runs on. The modular approach aims at fast prototyping with event-based sensors, enabling new actors in the field to test ideas or architectures even without knowing much about neuromorphic engineering or event-based processing.

The next decade will certainly see more and more of these sensors in consumer products. Hopefully, the spread of the sensors will push for the development of computation platforms

[1] It can be used on standard computers or embedded platforms.

adapted to them. The different systems, such as the SpiNNaker platform, that we have been able to use during my Ph.D. are only at the early stages of a new kind of computation.

The data representation offered by neuromorphic sensors is very well suited for distributed, massively parallel architectures. With the speed of current electronic devices, coding information in time is a very promising possibility. High temporal resolution of devices allows for a trade-off between the accuracy of computation and the time needed to obtain a result to stay in relevant boundaries for concrete applications.

We can imagine a system, computing very quickly a first approximation of a task while correcting it later on with a much more precise result without changing its core architecture and without having to accumulate and integrate approximations for a long period of time.

The STICK framework we propose in this work is offering a real alternative to standard computers. Most of today's highly parallel machines are based on the same computational principles as standard computers. They are just adding more and more processors together with smart architectures allowing for better communication in-between these units. But they always face the same problem: memory. Whatever the algorithm they try to implement, it has to work on data and parameters. Accessing this data and the parameters of the algorithm then becomes a huge bottleneck known as the *Von Neumann* bottleneck.

This is why we propose a radically different approach. By merging computation and memory, there is no longer any need to move data between storage spaces and computing spaces. This spares a lot of time and energy for actual computation and saves resources.

Implementing this framework on standard processors does not make sense from a purely computational point of view but I believe that its full potential can be unlocked by designing a mixed-analog-digital system where its neuron-like units are implemented in analog circuitry with an underlying fabric of digital communication.

Another question that this framework raises is its biological plausibility. When designing its units, care has been taken to stay as close as possible from biological behaviors and to make use of processes which can be found in real neurons. In the interest of developing a possible next generation of computers, it is best to go beyond the limitations that biology imposes to us like the maximum time precision of events in the system. But going back to this neural inspiration would be of great interest.

Appendix A

List of publications

A.1 Conference papers

1. **Event-based features for robotic vision.** Xavier Lagorce, Sio-Hoi Ieng, and Ryad Benosman. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on.*

2. **Accelerated frame-free time-encoded multi-step imaging.** Garrick Orchard, Daniel Matolin, Xavier Lagorce, Ryad Benosman, and Christoph Posch. In *Circuits And Systems (ISCAS), 2014 IEEE Internation Symposium on.*

3. **Real-time Event-driven Spiking Neural Network Object Recognition on the SpiNNaker Platform.** Garrick Orchard, Xavier Lagorce, Christoph Posch, Ryad Benosman, and Francesco Galluppi. In *Circuits And Systems (ISCAS), 2015 IEEE Internation Symposium on.*

A.2 Journal papers

1. **Event-based visual flow.** Ryad Benosman, Charles Clercq, Xavier Lagorce, Sio-Hoi Ieng, and Chiara Bartolozzi. In *IEEE Transactions on Neural Networks and Learning Systems.* (2014)

2. **Asynchronous event-based multikernel algorithm for high-speed visual features tracking.** Xavier Lagorce, Cédric Meyer, Sio-Hoi Ieng, David Filliat, and Ryad Benosman. In *IEEE Transactions on Neural Networks and Learning Systems.* (2014)

3. **A framework for plasticity implementation on the SpiNNaker neural architecture.** Francesco Galluppi, Xavier Lagorce, Evangelos Stromatias, Michael Pfeiffer, Luis A Plana, Steve B Furber, and Ryad B Benosman. In *Frontiers in Neuroscience.* (2014)

4. **An asynchronous neuromorphic event-driven visual part-based shape tracking.** David Reverter Valeiras, Xavier Lagorce, Xavier Clady, Chiara Bartolozzi, Sio-Hoi Ieng, and Ryad Benosman. In *IEEE Transactions on Neural Networks and Learning Systems.* (2015)

5. **Spatiotemporal features for asynchronous event-based data.** Xavier Lagorce, Sio-Hoi Ieng, Xavier Clady, Michael Pfeiffer, and Ryad B Benosman. In *Frontiers in Neuroscience.* (2015)

6. **Breaking the millisecond barrier on spinnaker: Implementing asynchronous event-based plastic models with microsecond resolution.** Xavier Lagorce, Evangelos Stromatias, Francesco Galluppi, Luis A Plana, Shih-Chii Liu, Steve B Furber, and Ryad Benjamin Benosman. In *Frontiers in Neuroscience.* (2015)

7. **Increased display frame rate improves visual performance for moving stimuli.** Sihem Kime, Francesco Galluppi, Xavier Lagorce, Ryad Benosman, and Jean Lorenceau. Under review in *Journal of Vision.* (2015)

8. **HOTS: A Hierarchy Of event-based Time-Surfaces for pattern recognition.** Xavier Lagorce, Garrick Orchard, Francesco Galluppi, Bertram Shi, and Ryad Benosman. Submitted to *IEEE Transactions on Pattern Analysis and Machine Intelligence.* (2015)

9. **Time-Oriented Computation of Motion Flow on Neuromorphic Platform.** Massimiliano Giulioni, Xavier Lagorce, Francesco Galluppi, and Ryad Benosman. Submitted to *Frontiers in Neuroscience.* (2015)

10. **STICK: Spike Time Interval Computational Kernel.** Xavier Lagorce, and Ryad Benosman. Submitted to *Neural Computation.* (2015)

11. **Neuromorphic Event-based Stereovision using Time Surface.** Sio-Hoi Ieng, João Carneiro, Marc Oswald, Xavier Lagorce, and Ryad Benosman. In prepration. (2015)

Appendix B

List of patents

1. **Dispositif de traitement de données avec représentation de valeurs par des intervalles de temps entre événements.**
   Xavier Lagorce, Ryad Benosman. [Submitted]

2. **Detection de formes.**
   Xavier Lagorce, Bertram Shi, Ryad Benosman. [Submitted]

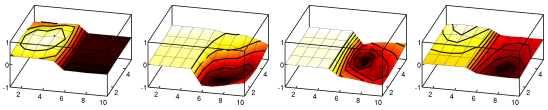
3. **Stereo matching.**
   Sio-Hoi Ieng, Xavier Lagorce, Ryad Benosman. [Submitted]

4. **Downsampling for retina camera.**
   Francesco Galluppi, Xavier Lagorce, Guillaume Chenegros, Ryad Benosman. [In preparation]

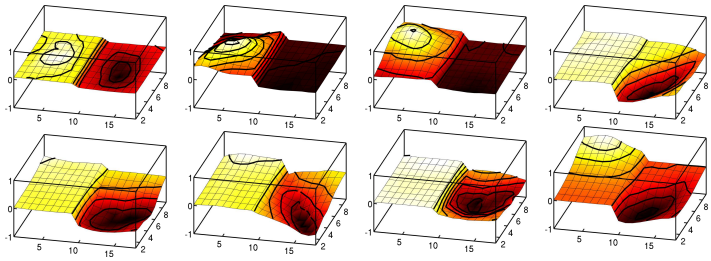
Appendix C

HOTS: Supplemental figures

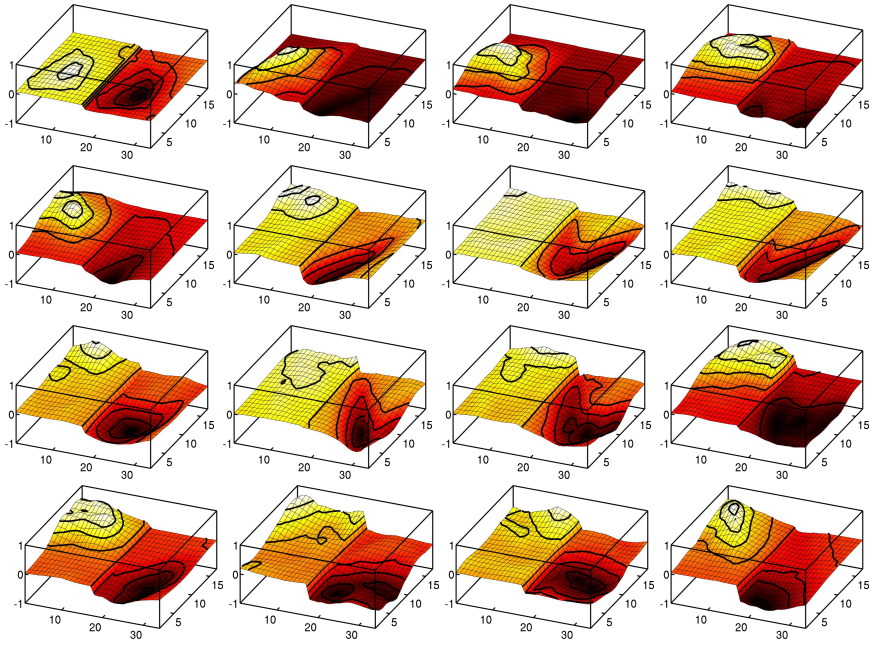
This appendix regroups some detailed figures about the experiments contained in Chapter 3. Fig. C.1 and Fig. C.2 presents the features learnt by the three layers of our architecture for the flipped cards and letters & digits experiments respectively. Fig. C.3 and Fig. C.4 show the signatures obtained for all the characters in the letters & digits task. Finally, Fig. C.5 and Fig. C.6 present the obtained for the face recognition experiment.



(a) Layer 1

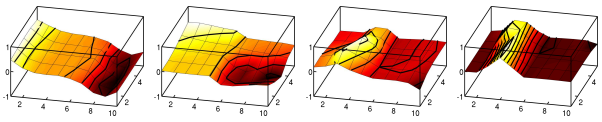


(b) Layer 2

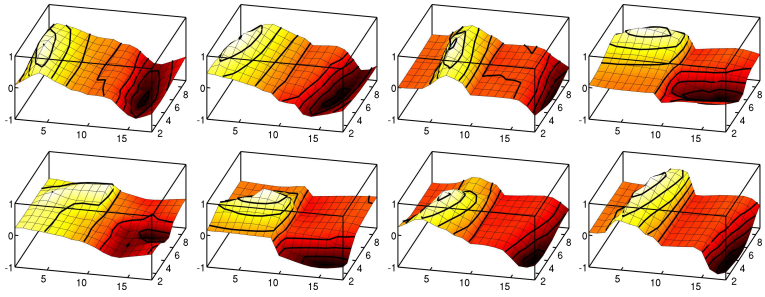


(c) Layer 3

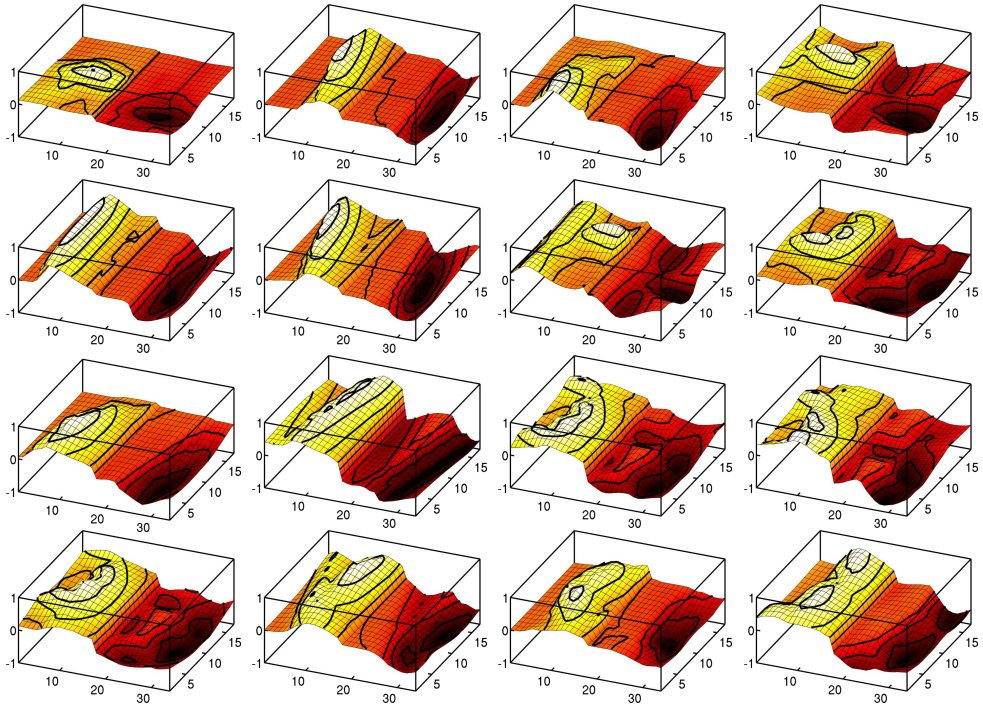
Figure C.1: Flipped cards experiment: Reconstructed features extracted by the different layers. This figure presents the obtained cluster centers for the three successive layers used in the cards recognition task (4 features for layer 1, 8 for layer 2, 16 for layer 3). The first layer is directly obtained from events generated by the event-driven time-based vision sensor. Each feature is represented as a time-surface like in Fig. 3.2, the first, positive, half corresponds to the ON events and the second, negative, half corresponds to the OFF events. Each layer is then using the features of its preceding layer to interpret its own features as a pattern of event produced by the camera. As the position of the layer increases, the size of the feature increases. So does its complexity. The actual features used by the classifier are the features from layer 3.



(a) Layer 1



(b) Layer 2



(c) Layer 3

Figure C.2: Letters & Digits experiment: Reconstructed features extracted by the different layers.

This figure presents the obtained cluster centers for the three successive layers used in the letters & digits recognition task (4 features for layer 1, 8 for layer 2, 16 for layer 3). The first layer is directly obtained from events generated by the event-driven time-based vision sensor. Each feature is represented as a time-surface like in Fig. 3.2, the first, positive, half corresponds to the ON events and the second, negative, half corresponds to the OFF events. Each layer is then using the features of its preceding layer to interpret its own features as a pattern of event produced by the camera. As the position of the layer increases, the size of the feature increases. So does its complexity. The actual features used by the classifier are the features from layer 3.

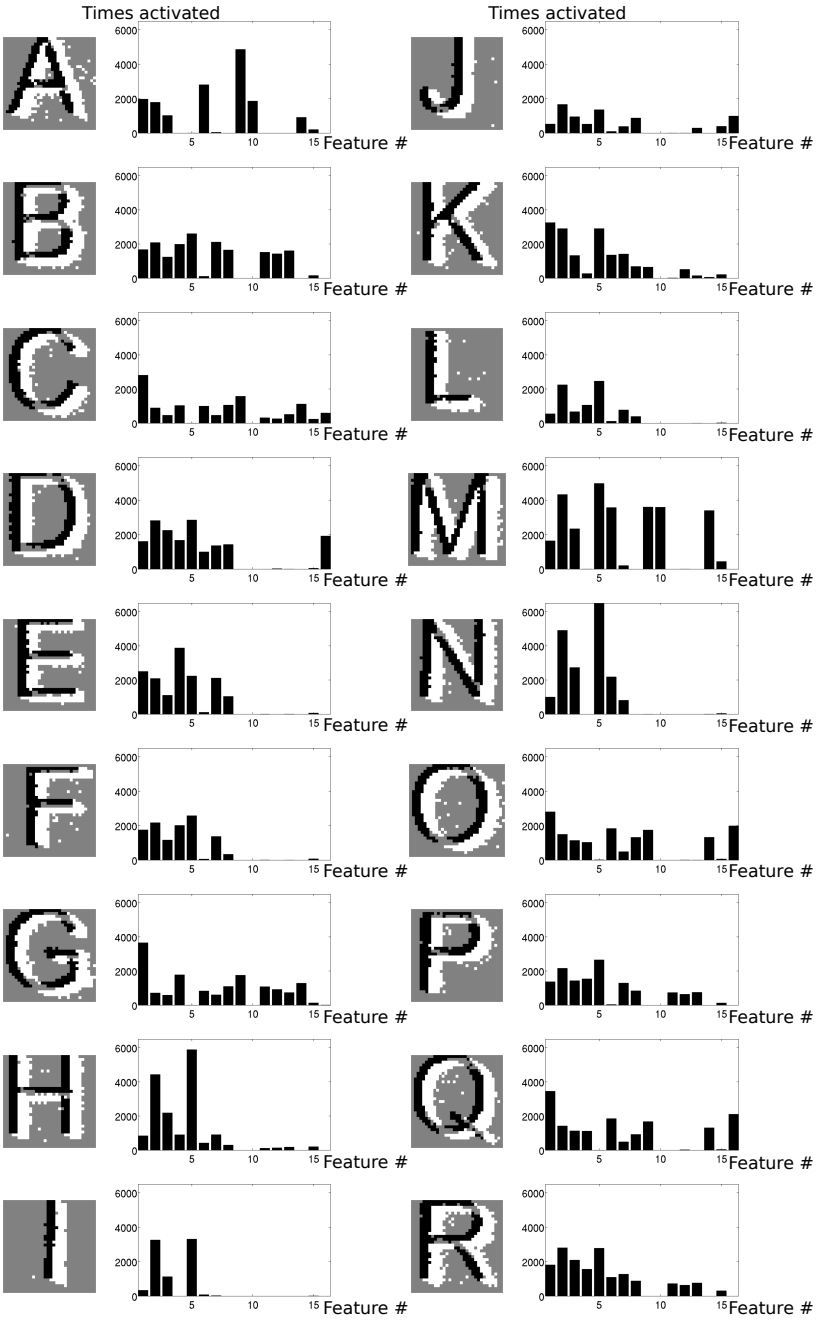


Figure C.3: Letters & Digits experiment: Pattern signatures (Letters from A to R).
For each letter and digit the trained histogram used as a signature by the classifier is shown. The snapshot shows an accumulation of events from the sensor (White dots for ON events and black dots for OFF events). The histograms present the signatures: X axis is the index of the feature shown in Fig. C.2, Y axis is the number of activations of the feature during the stimulus presentation.

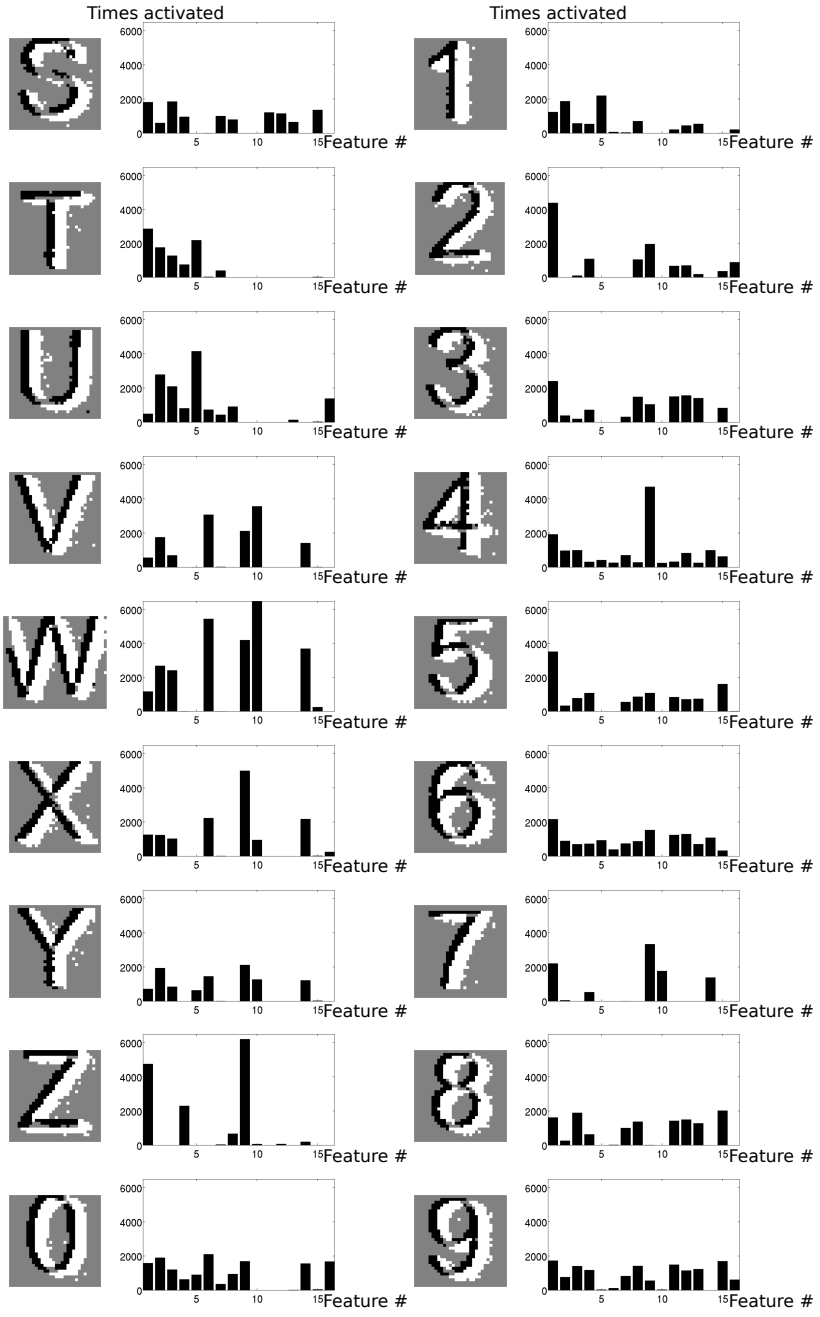
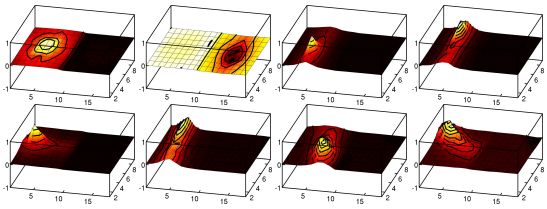
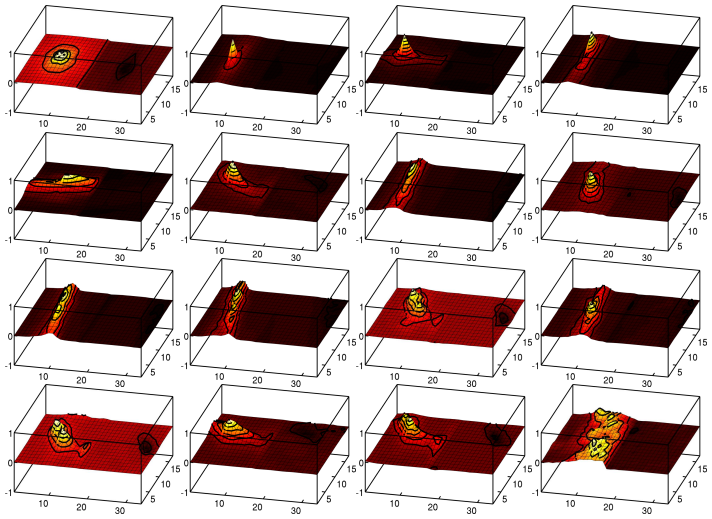


Figure C.4: Letters & Digits experiment: Pattern signatures (Letters from S to Z and digits). For each letter and digit the trained histogram used as a signature by the classifier is shown. The snapshot shows an accumulation of events from the sensor (White dots for ON events and black dots for OFF events). The histograms present the signatures: X axis is the index of the feature shown in Fig. C.2, Y axis is the number of activations of the feature during the stimulus presentation.



(a) Layer 1



(b) Layer 2

Figure C.5: Face recognition experiment: Reconstructed features extracted by layers 1 and 2. This figure presents the obtained cluster centers for the first two layers used in the face recognition task (8 features for layer 1, 16 for layer 2, 32 for layer 3). The first layer is directly obtained from events generated by the event-driven time-based vision sensor. Each feature is represented as a time-surface like in Fig. 3.2, the first, positive, half corresponds to the ON events and the second, negative, half corresponds to the OFF events. Each layer is then using the features of its preceding layer to interpret its own features as a pattern of event produced by the camera. As the position of the layer increases, the size of the feature increases. So does its complexity. See Fig. C.6 for the third layer.

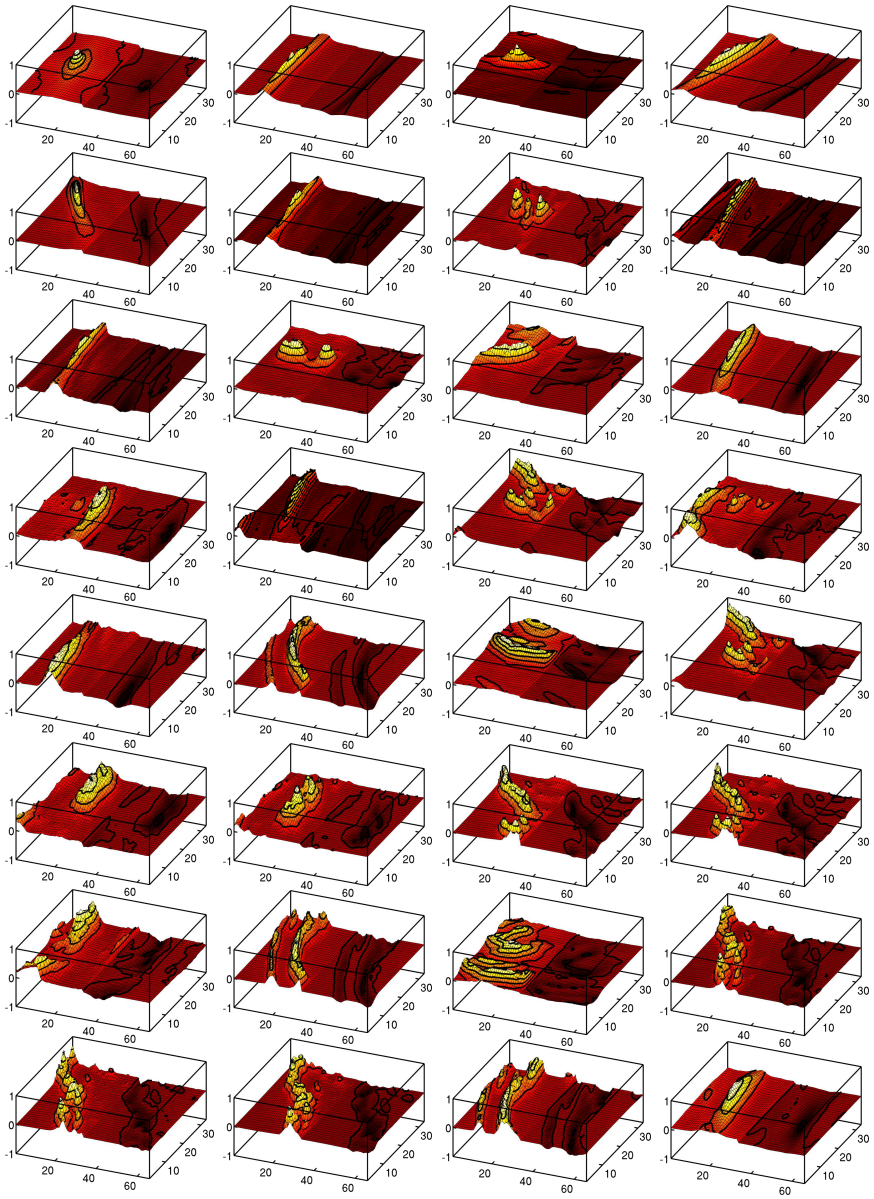


Figure C.6: Face recognition experiment: Reconstructed features extracted by layer 3. This figure presents the obtained cluster centers for the third and last layer used in the face recognition task (8 features for layer 1, 16 for layer 2, 32 for layer 3). The first layer is directly obtained from events generated by the event-driven time-based vision sensor. Each feature is represented as a time-surface like in Fig. 3.2, the first, positive, half corresponds to the ON events and the second, negative, half corresponds to the OFF events. Each layer is then using the features of its preceding layer to interpret its own features as a pattern of event produced by the camera. As the position of the layer increases, the size of the feature increases. So does its complexity. The actual features used by the classifier are the features from layer 3. See Fig. C.5 for the first two layers.

Appendix D

STICK: Chronograms and detailled proofs

This appendix presents the detailed proofs of all the networks described in Section 6.2

D.1 Storing data: analog memories

D.1.1 Inverting Memory

The Inverting Memory network (see Fig. 6.2) receives 2 spikes on the *input* neuron at times t^1_{in} and t^2_{in} such that $\Delta T_{in} = t^2_{\text{in}} - t^1_{\text{in}}$ encodes its input value. When *input* spikes at t^1_{in}, synaptic connections are activated towards neurons *first* and *last*. Because of the synaptic delays and weights, *last*'s membrane potential is set to $V_t/2$ and *first* spikes at time $t^1_{\text{first}} = t^1_{\text{in}} + T_{\text{syn}} + T_{\text{neu}}$, T_{neu} being the time needed by *first* to produce a spike. When *first* spikes, an inhibitory connection to itself sets its potential to $-V_t$ while a second connection triggers the integration of neuron *acc* after a delay $T_{\text{syn}} + T_{\text{min}}$ with weight w_{acc}. We then have:

$$\begin{aligned} t^1_{\text{st}} &= t^1_{\text{first}} + T_{\text{syn}} + T_{\text{min}} && \text{(D.1)} \\ &= t^1_{\text{in}} + 2.T_{\text{syn}} + T_{\text{neu}} + T_{\text{min}}. && \text{(D.2)} \end{aligned}$$

When *input* spikes for the second time at time t^2_{in}, *first*'s membrane potential gets back to its reset value while *last* reaches its threshold. This produces a spike from neuron *last* at time $t^1_{\text{last}} = t^2_{\text{in}} + T_{\text{syn}} + T_{\text{neu}}$. The connection to *acc* with delay T_{syn} and weight $-w_{\text{acc}}$ stops *acc*'s integration at time:

$$\begin{aligned} t^1_{\text{end}} &= t^1_{\text{last}} + T_{\text{syn}} && \text{(D.3)} \\ &= t^2_{\text{in}} + 2.T_{\text{syn}} + T_{\text{neu}}. && \text{(D.4)} \end{aligned}$$

During this integration window we had, for neuron *acc*, $g_e = w_{\text{acc}}$ such that, the membrane potential of *acc* at time t^1_{end} is:

$$\begin{aligned} V_{sto} &= \frac{w_{\text{acc}}}{\tau_m}.(t^1_{\text{end}} - t^1_{\text{st}}) && \text{(D.5)} \\ &= \frac{w_{\text{acc}}}{\tau_m}.(t^2_{\text{in}} - (t^1_{\text{in}} + T_{\text{min}})) && \text{(D.6)} \\ &= \frac{w_{\text{acc}}}{\tau_m}.(\Delta T_{in} - T_{\text{min}}) && \text{(D.7)} \end{aligned}$$

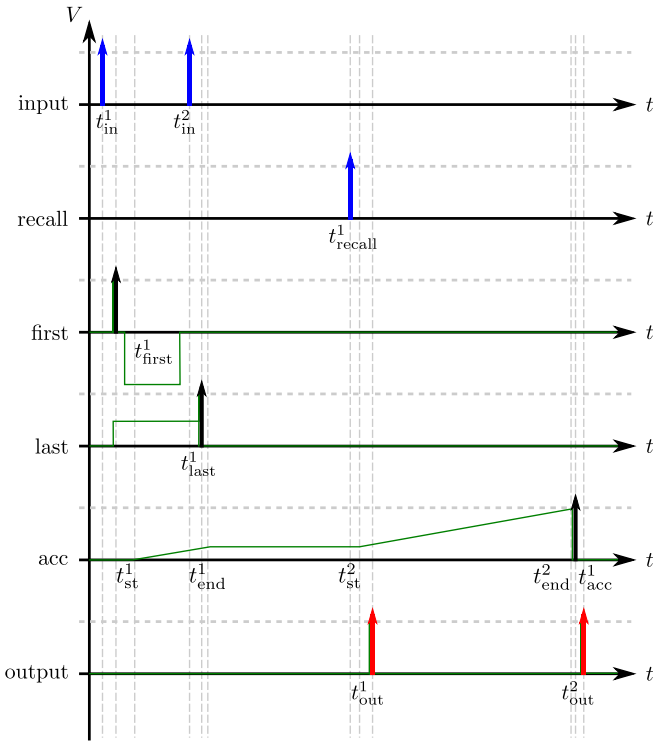


Figure D.1: Inverting Memory: chronogram of the network for an input at times t^1_{in} and t^2_{in} and a recall at time t^1_{recall}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neurons, recall of Fig. 6.3)

When the *recall* neuron receives an input at time t^1_{recall}, *acc*'s integration starts again at time $t^2_{\text{st}} = t^1_{\text{recall}} + T_{\text{syn}}$. At the same time, its second connection triggers a spike of the *output* neuron at time:

$$t^1_{\text{out}} = t^1_{\text{recall}} + (2.T_{\text{syn}} + T_{\text{neu}}) + T_{\text{neu}} \quad \text{(D.8)}$$

$$= t^1_{\text{recall}} + 2.T_{\text{syn}} + 2.T_{\text{neu}} \quad \text{(D.9)}$$

The integration stops again when *acc* reaches its threshold at time t^2_{end}, giving the following equation:

$$V_t = \frac{w_{\text{acc}}}{\tau_m}.(t^2_{\text{end}} - t^2_{\text{st}}) + V_{sto} \quad \text{(D.10)}$$

$$t^2_{\text{end}} = (t^1_{\text{recall}} + T_{\text{syn}}) - (\Delta T_{in} - T_{\text{min}}) + V_t.\frac{\tau_m}{w_{\text{acc}}}, \quad \text{(D.11)}$$

By definition of w_{acc}, we have $V_t.\tau_m/w_{\text{acc}} = T_{\text{max}}, so$:

$$t^2_{\text{end}} = t^1_{\text{recall}} + T_{\text{syn}} + T_{\text{max}} - (\Delta T_{in} - T_{\text{min}}), \quad \text{(D.12)}$$

Because *acc1* then needs the time T_{neu} to produce a spike, we get:

$$t^1_{\text{acc}} = t^2_{\text{end}} + T_{\text{neu}} \quad \text{(D.13)}$$

We thus get the second spike of *output* at time:

$$t^2_{\text{out}} = t^1_{\text{acc1}} + T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.14)}$$

$$t^2_{\text{out}} = t^1_{\text{recall}} + 2.T_{\text{syn}} + 2.T_{\text{neu}} + T_{\text{max}} - (\Delta T_{in} - T_{\text{min}}) \quad \text{(D.15)}$$

such that:

$$\Delta T_{out} = t^2_{\text{out}} - t^1_{\text{out}} \quad \text{(D.16)}$$

$$= T_{\text{max}} - (\Delta T_{in} - T_{\text{min}}). \quad \text{(D.17)}$$

D.1.2 Memory

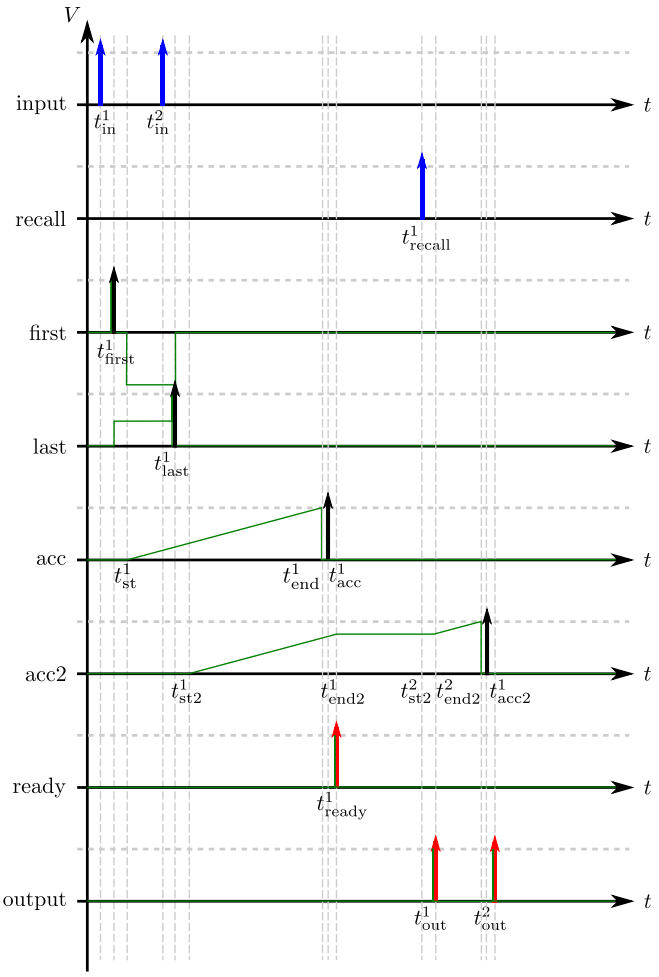


Figure D.2: Memory: chronogram of the network for an input at times t^1_{in} and t^2_{in} and a recall at time t^1_{recall}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neuron.)

The Memory network (see Fig. 6.4) receives 2 spikes on the *input* neuron at times t^1_{in} and t^2_{in} such that $\Delta T_{in} = t^2_{\text{in}} - t^1_{\text{in}}$ encodes its input value. With the reasoning used in the previous

subsection, we get:

$$t^1_{\text{first}} = t^1_{\text{in}} + T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.18)}$$
$$t^1_{\text{last}} = t^2_{\text{in}} + T_{\text{syn}} + T_{\text{neu}}. \quad \text{(D.19)}$$

When *first* spikes, it triggers integration in *acc*'s membrane potential. Because of the synaptic delay, we get:

$$t^1_{\text{st}} = t^1_{\text{first}} + T_{\text{syn}} = t^1_{\text{in}} + 2.T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.20)}$$

Neuron *acc* continues to integrate its w_{acc} input until reaching its threshold. This gives us the following equation:

$$V_t = \frac{w_{\text{acc}}}{\tau_m}.(t^1_{\text{end}} - t^1_{\text{st}}) \quad \text{(D.21)}$$

by definition of w_{acc}, we thus get:

$$t^1_{\text{end}} = t^1_{\text{st}} + T_{\text{max}} \quad \text{(D.22)}$$
$$= t^1_{\text{in}} + T_{\text{max}} + 2.T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.23)}$$
$$t^1_{\text{acc}} = t^1_{\text{end}} + T_{\text{neu}} \quad \text{(D.24)}$$
$$= t^1_{\text{in}} + T_{\text{max}} + 2.T_{\text{syn}} + 2.T_{\text{neu}} \quad \text{(D.25)}$$
$$t^1_{\text{end2}} = t^1_{\text{acc}} + T_{\text{syn}} \quad \text{(D.26)}$$
$$= t^1_{\text{in}} + T_{\text{max}} + 3.T_{\text{syn}} + 2.T_{\text{neu}} \quad \text{(D.27)}$$

At the same time, integration starts in *acc2*'s membrane potential after *last* spikes, which corresponds to:

$$t^1_{\text{st2}} = t^1_{\text{last}} + T_{\text{syn}} = t^2_{\text{in}} + 2.T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.28)}$$

The membrane potential of *acc2* after the end of the first integration phase is thus:

$$V_{sto} = \frac{w_{\text{acc}}}{\tau_m}.(t^1_{\text{end2}} - t^1_{\text{st2}}) \quad \text{(D.29)}$$
$$= \frac{w_{\text{acc}}}{\tau_m}.(T_{\text{max}} + t^1_{\text{in}} - t^2_{\text{in}} + T_{\text{syn}} + T_{\text{neu}}) \quad \text{(D.30)}$$
$$= \frac{w_{\text{acc}}}{\tau_m}.(T_{\text{max}} - \Delta T_{in} + T_{\text{syn}} + T_{\text{neu}}) \quad \text{(D.31)}$$

When the *recall* neuron is triggered, it starts *acc2*'s integration again at time:

$$t^2_{\text{st2}} = t^1_{\text{recall}} + T_{\text{syn}} \quad \text{(D.32)}$$

Which then produces the first output spike at time:

$$t^1_{\text{out}} = t^1_{\text{recall}} + T_{\text{syn}} + T_{\text{neu}}. \quad \text{(D.33)}$$

This second integration phase of *acc2* finishes when its threshold is reached:

$$V_t = \frac{w_{\text{acc}}}{\tau_m}.(t^2_{\text{end2}} - t^2_{\text{st2}}) + V_{sto} \quad \text{(D.34)}$$
$$T_{\text{max}} = t_{\text{end2}} - t^1_{\text{recall}} - T_{\text{syn}} + T_{\text{max}} - \Delta T_{in} + T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.35)}$$
$$t^2_{\text{end}} = t^1_{\text{recall}} + \Delta T_{in} - T_{\text{neu}} \quad \text{(D.36)}$$

We thus get a spike from *acc2* at time:

$$t^1_{\text{acc2}} = t^2_{\text{end2}} + T_{\text{neu}} = t^1_{\text{recall}} + \Delta T_{in} \tag{D.37}$$

leading to the second output spike at time:

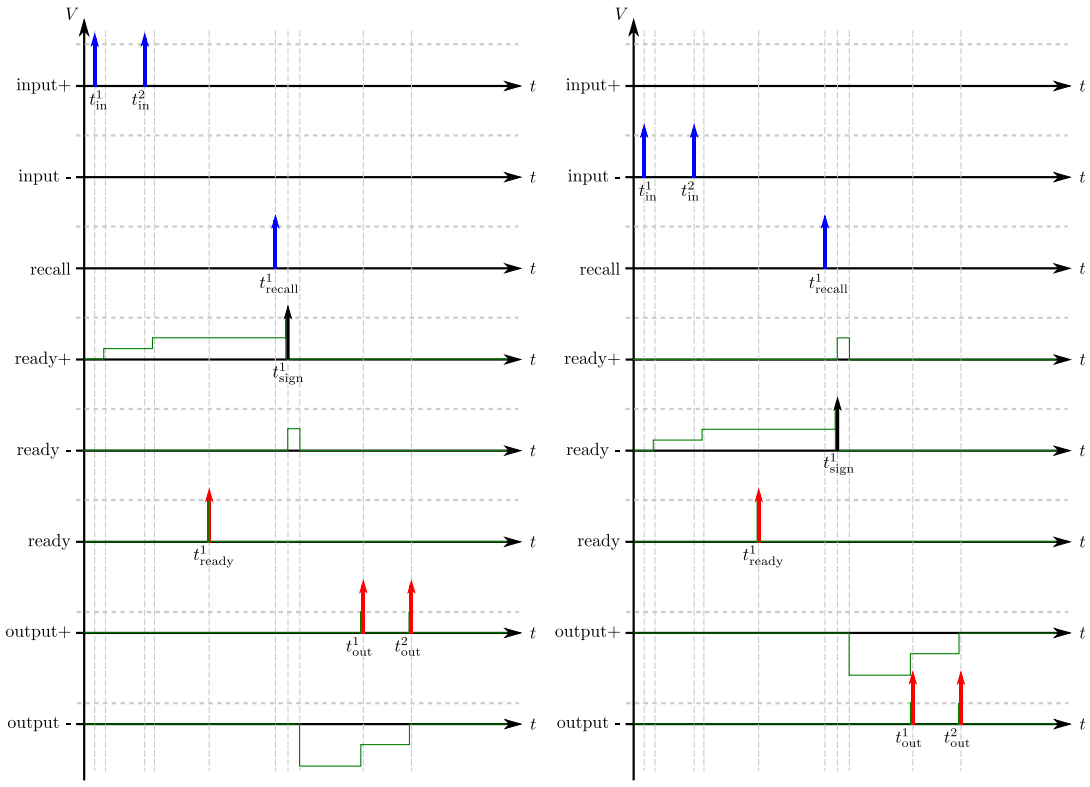
$$t^2_{\text{out}} = t_{\text{acc2}} + T_{\text{syn}} + T_{\text{neu}} = t^1_{\text{recall}} + \Delta T_{in} + T_{\text{syn}} + T_{\text{neu}} \tag{D.38}$$

such that:

$$\Delta T_{out} = t^2_{\text{out}} - t^1_{\text{out}} \tag{D.39}$$

$$= \Delta T_{in} \tag{D.40}$$

D.1.3 Signed Memory



(a) Positive input (b) Negative input

Figure D.3: Signed Memory: chronogram of the network for an input at times t^1_{in} and t^2_{in} and a recall at time t^1_{recall}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neurons). (a) Depicts the case of a positive input whereas (b) depicts the case of a negative input.

The Signed Memory network (see Fig. 6.5) receives 2 spikes on the *input* neuron at times t^1_{in} and t^2_{in} such that $\Delta T_{in} = t^2_{\text{in}} - t^1_{\text{in}}$ encodes its input value and the receiving neuron encodes the sign

of the input (*input* + for positive inputs, *input* - for negative inputs). Let's consider, without loss of generality, the case where the input is positive (Fig. D.3(a)). For each of the 2 input spikes, the *ready* + neuron receives a synaptic contribution of $0.25w_e$. When the input has been completely fed into the network, *ready* +'s membrane potential is thus resting at a value of $V_t/2$ while *ready* -'s is still resting at its reset potential. At the same time, the input spikes are fed into the central Memory network (see previous subsection) independently of its sign. When the value is stored in the Memory network, it outputs a spike on its *Rdy* output, which is propagated to the *ready* neuron.

When the *recall* neuron is triggered, its connections contribute to *ready* + and *ready* -'s membrane potentials with a weight of $0.5w_e$. This leads to a spike of the *ready* + neuron at time t^1_{sign} while *ready* -'s membrane potential moves to $V_t/2$. The lateral inhibition between the 2 *ready* neurons then sets *ready* - back to its reset potential. When *ready* + spikes, it triggers the recall of the Memory network and at the same time inhibits the output neuron corresponding to a negative value, *output* -, setting its potential to $-2.V_t$. When the Memory network outputs its stored value, spikes are transmitted to both output. Because of their respective potential at this moment, only the positive output *output* + spikes, while *output* -'s membrane potential is set back to its reset potential by the 2 output spikes of the Memory network.

Fig. D.3(b) shows the same principle applied to a negative input. One can notice that the *ready* + and *ready* - neurons are implementing a small state machine routing the spikes produced by the central Memory network to different output neurons depending on the input neurons.

D.1.4 Synchronizer

The Synchronizer network (see Fig. 6.6) for N inputs uses N Memory networks. Fig. D.4 presents the chronogram of this network for $N = 2$. Every time an internal memory has finished storing an input, its *Rdy* output activates the *sync* neuron with a weight w_e/N. Thus, after i inputs have been presented, *sync*'s membrane potential V^i_{sync} is:

$$V^i_{sync} = \frac{i}{N}.V_t. \tag{D.41}$$

The *sync* neuron thus spikes after the N^{th} and last memory is ready. It then recalls the values stored in all Memory networks at the same time, effectively synchronizing the first encoding spikes of all its outputs.

D.2 Relational operations

D.2.1 Minimum

The Minimum network (see Fig. 6.7) receives 2 different inputs (a pair of spikes) from each input neurons *input1* (t^1_{in1} and t^2_{in1}) and *input2* (t^1_{in2} and t^2_{in2}) such that $\Delta T_{in1} = t^2_{\text{in1}} - t^1_{\text{in1}}$ and $\Delta T_{in2} = t^2_{\text{in2}} - t^1_{\text{in2}}$ encode its 2 inputs. Let us consider, without loss of generality, the case where the first input (*input1*) is smaller than the second one as shown in Fig. D.5. Assuming the inputs to be synchronized, we have:

$$t^1_{\text{in1}} = t^1_{\text{in2}}, \tag{D.42}$$

When *input1* and *input2* are triggered by the first encoding spike of each input, *smaller1* and *smaller2*'s membrane potentials are set to $V_t/2$ and the *output* neuron spikes after a delay due to

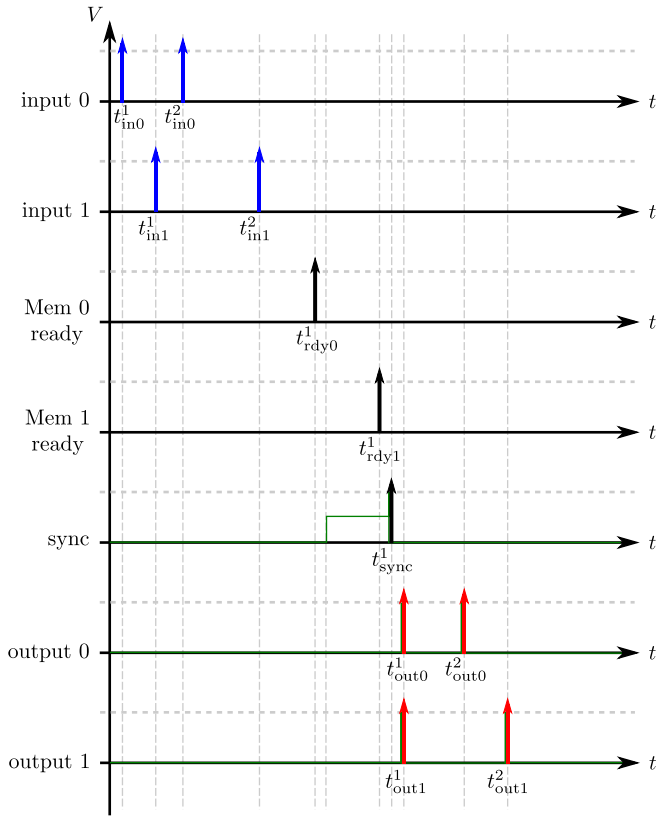


Figure D.4: Synchronizer: chronogram of the network for inputs at times t^1_{in0}, t^2_{in0} and t^1_{in1}, t^2_{in1}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neuron.)

the connection from *input1* and *input2* such that:

$$t_{\text{out1}} = t^1_{\text{in1}} + 2.T_{\text{syn}} + 2.T_{\text{neu}}. \tag{D.43}$$

When the smallest input (in this case *input1*) receives its second encoding spike at time t^2_{in1}, the *smaller1* neuron reaches its threshold and emits a spike at time:

$$t^1_{\text{smaller1}} = t^2_{\text{in1}} + T_{\text{syn}} + T_{\text{neu}}. \tag{D.44}$$

The spike from *smaller1* inhibits the other input (here *input2*) such that it will not be triggered by its second encoding spike (as can be seen in the chronogram). It also inhibits the *smaller2* neuron such that its membrane potential goes back to its reset potential. The second encoding spike from *input1* and the spike from *smaller1* are, in addition, enough contribution to trigger a second spike in the *output* neuron at time:

$$\begin{aligned} t^2_{\text{out}} &= \min\left\{t^2_{\text{in1}} + 2.T_{\text{syn}} + T_{\text{neu}}, t_{\text{smaller1}} + T_{\text{syn}}\right\} + T_{\text{neu}} && \text{(D.45)}\\ &= \min\left\{t^2_{\text{in1}} + 2.T_{\text{syn}} + T_{\text{neu}}, t^2_{\text{in1}} + 2.T_{\text{syn}} + T_{\text{neu}}\right\} + T_{\text{neu}} && \text{(D.46)}\\ &= t^2_{\text{in1}} + 2.T_{\text{syn}} + 2.T_{\text{neu}}. && \text{(D.47)} \end{aligned}$$

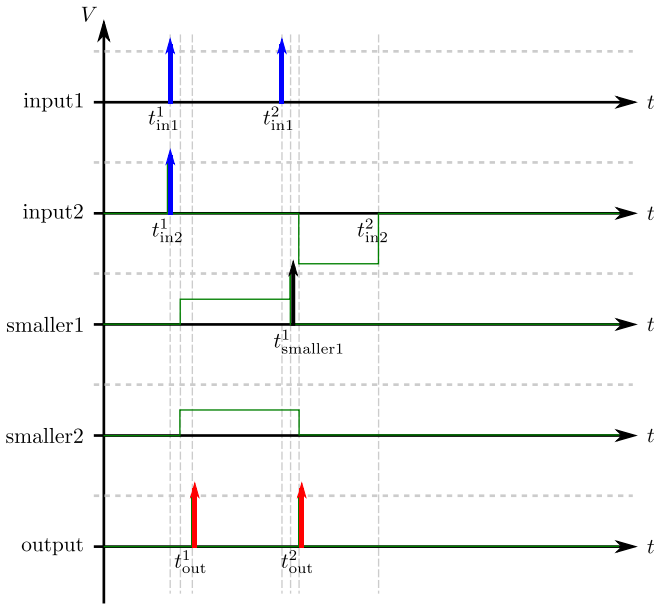


Figure D.5: Minimum: chronogram of the network for inputs at times t^1_{in1}, t^2_{in1} and t^1_{in2}, t^2_{in2}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neuron.)

We thus get an output pair of spikes spaced in time such that:

$$\Delta T_{out} = t^2_{\text{out}} - t^1_{\text{out}} = t^2_{\text{in1}} - t^1_{\text{in1}} = \Delta T_{in1}. \tag{D.48}$$

The output is thus corresponding to the smallest of the 2 inputs of the network while the indicator provides which of the two inputs is the smallest (*smaller1*). The same reasonning can be applied to the case where the second input (*input2*) is the smallest.

D.2.2 Maximum

The Maximum network (see Fig. 6.8) receives 2 different inputs (a pair of spikes) from each input neurons *input1* (t^1_{in1} and t^2_{in1}) and *input2* (t^1_{in2} and t^2_{in2}) such that $\Delta T_{in1} = t^2_{\text{in1}} - t^1_{\text{in1}}$ and $\Delta T_{in2} = t^2_{\text{in2}} - t^1_{\text{in2}}$ encode its 2 inputs. Let's consider, without loss of generality, the case where the second input (*input2*) is larger than the first one as depicted Fig. D.6. The 2 inputs being synchronized as a prerequisite of the network, we have:

$$t^1_{\text{in1}} = t^1_{\text{in2}}, \tag{D.49}$$

When *input1* and *input2* are triggered by the first encoding spike of each input, *larger1* and *larger2*'s membrane potentials are set to $V_t/2$ and the *output* neuron spikes after a delay due to the connection from *input1* and *input2* such that:

$$t_{\text{out1}} = t^1_{\text{in2}} + T_{\text{syn}} + T_{\text{neu}}. \tag{D.50}$$

When the smallest input (in this case *input1*) receives its second encoding spike at time t^2_{in1}, we know that the other input has to be the larger one. Hence, the *larger2* neuron reaches its threshold

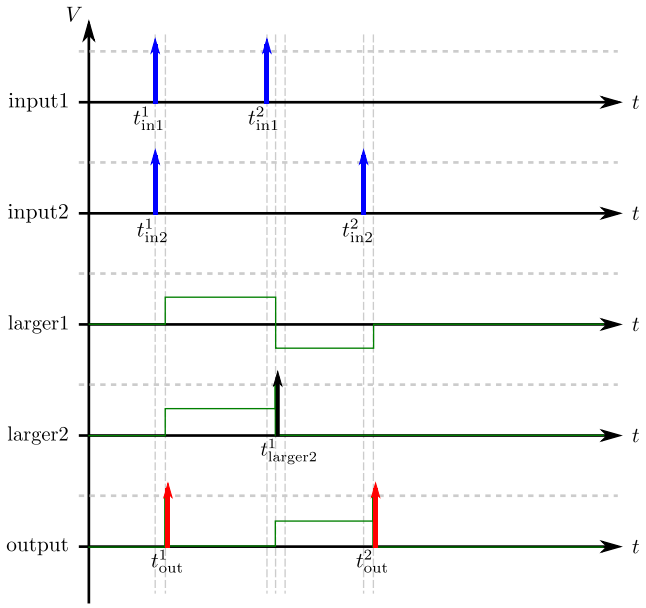


Figure D.6: Maximum: chronogram of the network for inputs at times t^1_{in1}, t^2_{in1} and t^1_{in2}, t^2_{in2}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neuron.)

and emits a spike at time:

$$t^1_{\text{larger2}} = t^2_{\text{in1}} + T_{\text{syn}} + T_{\text{neu}}. \tag{D.51}$$

The spike from *larger2* inhibits the *larger1* neuron such that its membrane potential goes down to $-V_t/2$, while the connection *input1* to *output* raises *output*'s membrane potential to $V_t/2$. When the second encoding spike from *input2* is received, the connection from *input2* to *larger1* moves back *larger1*'s membrane potential to its reset potential while its connection to *output* triggers a spike at time:

$$t^2_{\text{out}} = t^2_{\text{in2}} + T_{\text{syn}} + T_{\text{neu}}. \tag{D.52}$$

We thus get an output pair of spikes spaced in time such that:

$$\Delta T_{out} = t^2_{\text{out}} - t^1_{\text{out}} = t^2_{\text{in2}} - t^1_{\text{in2}} = \Delta T_{in2}. \tag{D.53}$$

The output is thus corresponding to the largest of the 2 inputs of the network.while the indicator provides which of the two inputs is the largest (*larger1*). The same reasonning can be applied to the case where the first input (*input1*) is the largest.

D.3 Linear operations

D.3.1 Subtractor

The Subtractor network (see Fig. 6.9) receives 2 different inputs (a pair of spikes) from each input neurons *input1* (t^1_{in1} and t^2_{in1}) and *input2* (t^1_{in2} and t^2_{in2}) such that $\Delta T_{in1} = t^2_{\text{in1}} - t^1_{\text{in1}}$ and $\Delta T_{in2} = t^2_{\text{in2}} - t^1_{\text{in2}}$ encode its 2 inputs. Let us consider, without loss of generality, the case where the first

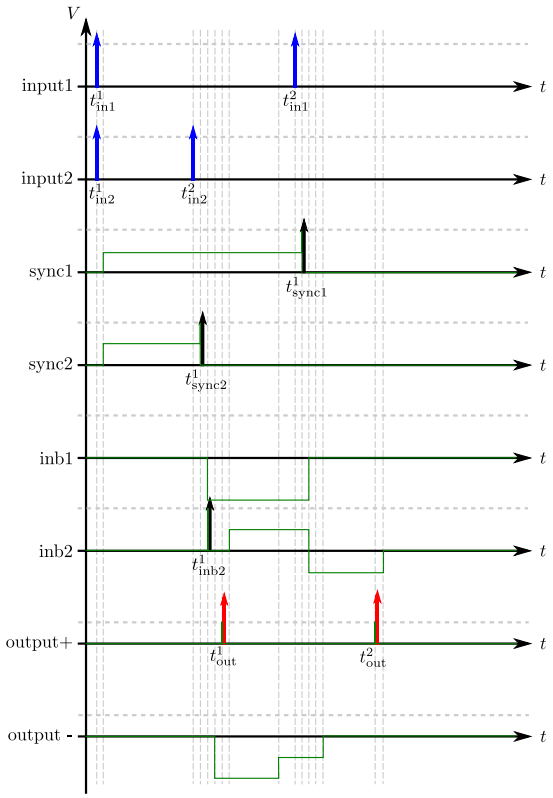


Figure D.7: Subtractor: chronogram of the network for inputs at times t^1_{in1}, t^2_{in1} and t^1_{in2}, t^2_{in2}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neuron.)

input (*input1*) is larger than the second one as depicted Fig. D.7. Assuming the inputs to be synchronized, we have:

$$t^1_{in1} = t^1_{in2}. \tag{D.54}$$

When *input1* and *input2* are triggered by the first encoding spikes of each input, they activate the *sync1* and *sync2* neurons such that their membrane potentials are now set to $V_t/2$. When the second encoding spike of the smallest input (here, *input2*) is received, the activation from *input2* to *sync2* is sufficient to trigger a spike at time:

$$t^1_{sync2} = t^2_{in2} + T_{syn} + T_{neu}. \tag{D.55}$$

This spike inhibits the *inb1* neuron after a time T_{syn}, moving its membrane potential to $-V_t$ and, because the sign of the output is now known, it triggers and output spikes on *output* + at time:

$$t^1_{out} = T^1_{sync2} + 3.T_{syn} + 2.T_{neu} + T_{neu} = T^2_{in2} + 4.T_{syn} + 4.T_{neu}. \tag{D.56}$$

It also contributes to *output* -'s membrane potential with an activation of w_e at time $t^1_{sync2} + T_{min} + 3.T_{syn} + 2.T_{neu}$. But before this contribution reaches *output* -, the *inb2* neuron is triggered and produces a spike at time:

$$t^1_{inb2} = t_{sync} + T_{syn} + T_{neu} \tag{D.57}$$

which inhibits *output* - with weight $2w_i$ at time $t^1_{\text{inb2}} + T_{\text{syn}} = t^1_{\text{sync2}} + T_{\text{syn}} + T_{\text{neu}}$. This inhibition thus happens before the direct excitation from *sync2*, which then leads to *output* - not emitting a spike and its membrane potential to be set to $-V_t$ after receiving these 2 spikes.

When the second encoding spike of the largest input is received, the spike from *input1* triggers a spike from *sync1* at time:

$$t^1_{\text{sync1}} = t^2_{\text{in1}} + T_{\text{syn}} + T_{\text{neu}}. \tag{D.58}$$

This spike leads to the inhibition of *inb2* to a membrane potential of $-V_t$ and an excitation of *inb1* back to its reset potential. It also activates *output* + back to its reset potential and triggers an output spike at time:

$$t^2_{\text{out}} = t_{\text{sync1}} + T_{\text{min}} + 3.T_{\text{syn}} + 2.T_{\text{neu}} + T_{\text{neu}} = t^2_{\text{in1}} + T_{\text{min}} + 4.T_{\text{syn}} + 4.T_{\text{neu}} \tag{D.59}$$

We thus get a positive output as expected and an output value:

$$\begin{aligned}
\Delta T_{out} &= t^2_{\text{out}} - t^1_{\text{out}} && \text{(D.60)}\\
&= T_{\text{min}} + t^2_{\text{in1}} - t^2_{\text{in2}} && \text{(D.61)}\\
&= T_{\text{min}} + (t^2_{\text{in1}} - t^1_{\text{in1}}) - (t^2_{\text{in2}} - t^1_{\text{in2}}) && \text{(D.62)}\\
&= T_{\text{min}} + (\Delta T_{in1} - \Delta T_{in2}) && \text{(D.63)}\\
&= T_{\text{min}} + (\Delta T_{in1} - T_{\text{min}}) - (\Delta T_{in2} - T_{\text{min}}). && \text{(D.64)}
\end{aligned}$$

Knowing that for each value x, we encode it as a time interval $f(x) = T_{\text{min}} + x.T_{\text{cod}}$, we have:

$$\begin{aligned}
\Delta T_{out} &= T_{\text{min}} + x_{in1}.T_{\text{cod}} - x_{in2}.T_{\text{cod}} && \text{(D.65)}\\
&= T_{\text{min}} + (x_{in1} - x_{in2}).T_{\text{cod}} && \text{(D.66)}
\end{aligned}$$

which corresponds to the encoding of the result $input1 - input2$.

D.3.2 Linear Combination

The first part of the Linear Combination presented Fig. 6.11 can be decomposed in a series of simpler circuits to what has been presented before for the inverting memory. For each of the N inputs, one branch is managing positive inputs while the other one is managing negative inputs (only one of these 2 can be activated in any computation). The architecture of each of these branches can be seen as an inverting memory (see Fig. 6.2) storing a value either in *acc1+* or *acc1-*. The targeted accumulator is chosen depending on the sign of the input and the sign of its associated coefficient α_i to represent the sign of the input's contribution to the result. *acc1+* is accumulating all positive contributions (positive inputs with positive coefficients and negative inputs with negative inputs) and *acc1-* is accumulating all negative contributions (negative inputs with positive coefficients and positive inputs with negative coefficients). With the same reasoning leading to Eq. (D.7) we can get the contribution of an input i to its associated accumulator, if ΔT^i_{in} is the input interspike associated to the considered input:

$$V^i_{sto} = |\alpha_i|.\frac{w_{\text{acc}}}{\tau_m}.(\Delta T^i_{in} - T_{\text{min}}), \tag{D.67}$$

The integration in the accumulators being linear, we obtain the membrane potential stored in the 2 accumulators after all inputs have been fed into the network:

$$V_{sto}^{acc1+} = \sum_{i\in I^+} V_{sto}^i \tag{D.68}$$

$$= \sum_{i\in I^+} |\alpha_i|.\frac{w_{\text{acc}}}{\tau_m}.(\Delta T_{in}^i - T_{\min}) \tag{D.69}$$

$$V_{sto}^{acc1-} = \sum_{i\in I^-} V_{sto}^i \tag{D.70}$$

$$= \sum_{i\in I^-} |\alpha_i|.\frac{w_{\text{acc}}}{\tau_m}.(\Delta T_{in}^i - T_{\min}) \tag{D.71}$$

where I^+ is the set of inputs contributing positively to the output and I^- is the set of inputs contributing negatively to the output. When the N inputs have been fed into the network, the *sync* neuron finally receives enough excitation to produce a spike at time t_{sync}^1. This spike triggers the readout process of *acc1+* and *acc1-* and, at the same time, starts integrating in neurons *acc2+* and *acc2-*. This process is similar to the one used in the Memory network (see Appendix D.1.2). If we consider the positive accumulator, we obtain spikes from *acc1+* and *acc2+* at time $t_{\text{acc1+}}^1$ and $t_{\text{acc2+}}^1$ respectively with the following conditions, where $t_{\text{st}}^1 = t_{\text{sync}}^1 + T_{\text{syn}}$ is the time at which the integration begins:

$$\begin{cases} V_t = \frac{w_{\text{acc}}}{\tau_m}.(t_{\text{acc1+}} - t_{\text{st}}^1) + V_{sto}^{acc1+} \\ V_t = \frac{w_{\text{acc}}}{\tau_m}.(t_{\text{acc2+}} - t_{\text{st}}^1) \end{cases} \tag{D.72}$$

By definition of w_{acc}, we thus get:

$$t_{\text{acc2+}} = t_{\text{st}}^1 + T_{\max} \tag{D.73}$$

and

$$T_{\max} = t_{\text{acc1+}} - t_{\text{st}}^1 + \sum_{i\in I^+} |\alpha_i|.(\Delta T_{in}^i - T_{\min}) \tag{D.74}$$

$$t_{\text{acc1+}} = t_{\text{st}}^1 + T_{\max} - \sum_{i\in I^+} |\alpha_i|.(\Delta T_{in}^i - T_{\min}) \tag{D.75}$$

Neuron *inter+* is thus producing 2 spikes with an interspike ΔT_{inter}^+ such that:

$$\Delta T_{inter}^+ = t_{\text{acc2+}} + T_{\min} + T_{\text{syn}} + T_{\text{neu}} - (t_{\text{acc1+}} + T_{\text{syn}} + T_{\text{neu}}) \tag{D.76}$$

$$= \sum_{i\in I^+} |\alpha_i|.(\Delta T_{in}^i - T_{\min}) + T_{\min} \tag{D.77}$$

The same reasoning on *acc1-* and *acc2-* leads to a pair of spikes on neuron *inter-* with an interspike ΔT_{inter}^-:

$$\Delta T_{inter}^- = \sum_{i\in I^-} |\alpha_i|.(\Delta T_{in}^i - T_{\min}) + T_{\min} \tag{D.78}$$

This two values are then synchronized by a Synchronizer network described previously and subtracted from one another such that the output is, according to Eq. (D.63):

$$\Delta T_{out} = \Delta T_{inter}^+ - \Delta T_{inter}^- + T_{\min} \tag{D.79}$$

$$= \sum_{i\in I^+} |\alpha_i|.(\Delta T_{in}^i - T_{\min}) - \sum_{i\in I^-} |\alpha_i|.(\Delta T_{in}^i - T_{\min}) + T_{\min} \tag{D.80}$$

$$= \sum_{i=0}^{N-1} \varepsilon_i.\alpha_i.(\Delta T_{in}^i - T_{\min}) + T_{\min} \tag{D.81}$$

where ε_i is $+1$ if input i is positive and -1 otherwise. This is the expected result of the linear combination.

D.4 Non-linear operations

D.4.1 Natural Logarithm

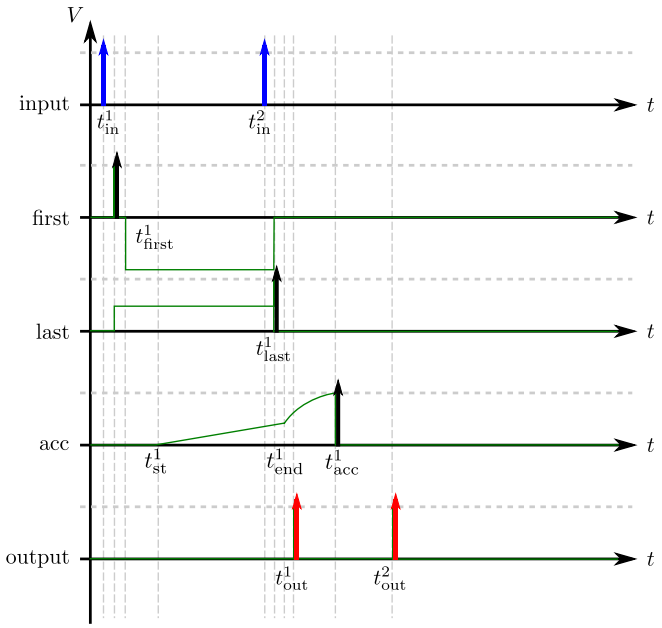


Figure D.8: Log: chronogram of the network for an input at times t^1_{in} and t^2_{in}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neurons).

The Log network (see Fig. 6.12) receives 2 spikes on the *input* neuron at times t^1_{in} and t^2_{in} such that $\Delta T_{in} = t^2_{\text{in}} - t^1_{\text{in}}$ encodes its input value. The same reasoning leads us to obtain the spike times of the *first* and *last* neurons at:

$$t^1_{\text{first}} = t^1_{\text{in}} + T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.82)}$$
$$t^1_{\text{last}} = t^2_{\text{in}} + T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.83)}$$

The *acc* neuron is thus integrating from time:

$$t^1_{\text{st}} = t^1_{\text{first}} + T_{\text{syn}} + T_{\text{min}} \quad \text{(D.84)}$$
$$= t^1_{\text{in}} + 2.T_{\text{syn}} + T_{\text{min}} + T_{\text{neu}} \quad \text{(D.85)}$$

to time:

$$t^1_{\text{end}} = t^1_{\text{last}} + T_{\text{syn}} \quad \text{(D.86)}$$
$$= t^2_{\text{in}} + 2.T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.87)}$$

The membrane potential of neuron *acc* at the end of this integration phase is thus:

$$V_{sto} = \frac{\bar{w}_{\text{acc}}}{\tau_m}(t^1_{\text{end}} - t^1_{\text{st}}) \quad \text{(D.88)}$$

$$= \frac{\bar{w}_{\text{acc}}}{\tau_m}(t^2_{\text{in}} - t^1_{\text{in}} - T_{\min}) \quad \text{(D.89)}$$

$$= \frac{\bar{w}_{\text{acc}}}{\tau_m}(\Delta T_{in} - T_{\min}) \quad \text{(D.90)}$$

If we name $\Delta T_{cod} = \Delta T_{in} - T_{\min}$ and considering the definition of $\bar{w}_{\text{acc}}$, we have:

$$V_{sto} = V_t.\frac{\Delta T_{cod}}{T_{\text{cod}}}. \quad \text{(D.91)}$$

The other synaptic connections from *last* to *acc* also activate the g_f dynamics of the *acc* neuron at time t^1_{end}. When the $g_f -$ synapse gets activated, *acc*'s membrane potential follows the following evolution obtained by solving the differential system Eq. (6.1):

$$V = V_{sto} + g_{\text{mult}}\frac{\tau_f}{\tau_m}(1 - e^{-(t-t^1_{\text{end}})/\tau_f}) \quad \text{(D.92)}$$

for $t \geq t^1_{\text{end}}$. According to Eq. (6.23), we chose $g_{\text{mult}} = V_t.\frac{\tau_m}{\tau_f}$ so that:

$$V = V_t.\frac{\Delta T_{cod}}{T_{\text{cod}}} + V_t.(1 - e^{-(t-t^1_{\text{end}})/\tau_f}) \quad \text{(D.93)}$$

The *acc* neuron will then spike at time t^1_{acc} when the condition $V = V_t$ is met. This gives us:

$$V_t = V_t.\frac{\Delta T_{cod}}{T_{\text{cod}}} + V_t.(1 - e^{-(t^1_{\text{acc}}-t^1_{\text{end}})/\tau_f}) \quad \text{(D.94)}$$

$$\frac{\Delta T_{cod}}{T_{\text{cod}}} = e^{-(t^1_{\text{acc}}-t^1_{\text{end}})/\tau_f} \quad \text{(D.95)}$$

$$t^1_{\text{acc}} = -\tau_f.\log\left(\frac{\Delta T_{cod}}{T_{\text{cod}}}\right) + t^1_{\text{end}} \quad \text{(D.96)}$$

The first output spike is generated by the connection from *last* to *output*, thus:

$$t^1_{\text{out}} = t^1_{\text{last}} + T_{\text{neu}} + 2.T_{\text{syn}} \quad \text{(D.97)}$$

While the second output spike is generated by the connection from *acc* to *output*, thus:

$$t^2_{\text{out}} = t^1_{\text{acc}} + T_{\text{neu}} + T_{\text{syn}} + T_{\min} \quad \text{(D.98)}$$

$$= -\tau_f.\log\left(\frac{\Delta T_{cod}}{T_{\text{cod}}}\right) + t^1_{\text{end}} + T_{\text{neu}} + T_{\text{syn}} + T_{\min} \quad \text{(D.99)}$$

$$= -\tau_f.\log\left(\frac{\Delta T_{cod}}{T_{\text{cod}}}\right) + t^1_{\text{last}} + T_{\text{neu}} + 2.T_{\text{syn}} + T_{\min} \quad \text{(D.100)}$$

This gives us the output ΔT_{out}:

$$\Delta T_{out} = t^2_{\text{out}} - t^1_{\text{out}} \quad \text{(D.101)}$$

$$= -\tau_f.\log\left(\frac{\Delta T_{cod}}{T_{\text{cod}}}\right) + T_{\min} \quad \text{(D.102)}$$

$$= T_{\min} + \tau_f.\log\left(\frac{T_{\text{cod}}}{\Delta T_{cod}}\right) \quad \text{(D.103)}$$

D.4.2 Exponential

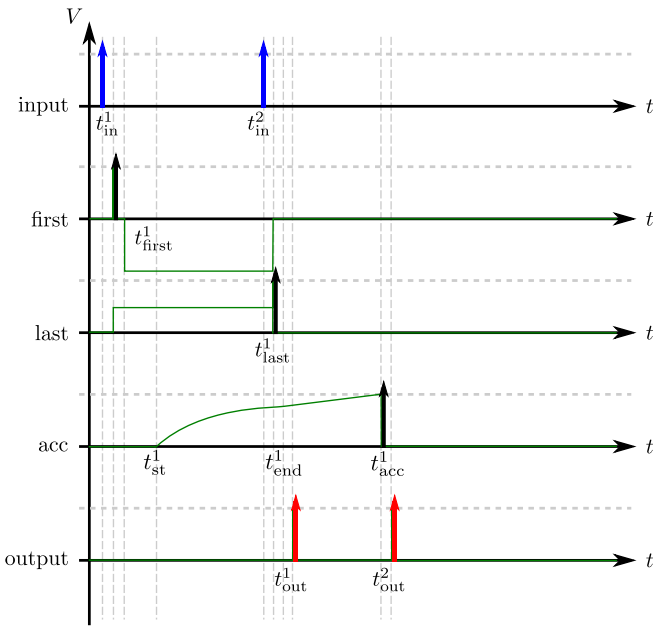


Figure D.9: Exp: chronogram of the network for an input at times t^1_{in} and t^2_{in}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neurons).

The Exp network (see Fig. 6.13) receives 2 spikes on the *input* neuron at times t^1_{in} and t^2_{in} such that $\Delta T_{in} = t^2_{\text{in}} - t^1_{\text{in}}$ encodes its input value. The same reasoning leads us to obtain the spike times of the *first* and *last* neurons at:

$$t^1_{\text{first}} = t^1_{\text{in}} + T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.104)}$$

$$t^1_{\text{last}} = t^2_{\text{in}} + T_{\text{syn}} + T_{\text{neu}} \quad \text{(D.105)}$$

The *first* neuron then triggers the g_f dynamics of neuron *acc* at time:

$$t^1_{\text{st}} = t^1_{\text{first}} + T_{\text{syn}} + T_{\text{min}} = t^1_{\text{in}} + T_{\text{neu}} + 2.T_{\text{syn}} + T_{\text{min}} \quad \text{(D.106)}$$

Solving the differential system Eq. (6.1), *acc*'s membrane potential is following the evolution:

$$V = g_{\text{mult}} \frac{\tau_f}{\tau_m} (1 - e^{-(t-t^1_{\text{st}})/\tau_f}) \quad \text{(D.107)}$$

$$= V_t.(1 - e^{-(t-t^1_{\text{st}})/\tau_f}) \quad \text{(D.108)}$$

This evolution is stopped at t^1_{end} by the connection from *last* to *acc* through its action on the *gate* signal:

$$t^1_{\text{end}} = t^1_{\text{last}} + T_{\text{syn}} = t^2_{\text{in}} + T_{\text{neu}} + 2.T_{\text{syn}} \quad \text{(D.109)}$$

At the end of this phase, *acc*'s membrane potential is thus equal to:

$$\begin{aligned} V_{sto} &= V_t.(1-e^{-(t^1_{\text{end}}-t^1_{\text{st}})/\tau_f}) && \text{(D.110)}\\ &= V_t.(1-e^{-(t^2_{\text{in}}-t^1_{\text{in}}-T_{\min})/\tau_f}) && \text{(D.111)}\\ &= V_t.(1-e^{-(\Delta T_{in}-T_{\min})/\tau_f}) && \text{(D.112)}\\ &= V_t.(1-e^{-\Delta T_{cod}/\tau_f}) && \text{(D.113)} \end{aligned}$$

with $\Delta T_{cod} = \Delta T_{in} - T_{\min}$. At the same time, *last* is starting a second integration process of a $g_e -$ synapse. *acc*'s membrane potential is then following the evolution:

$$V = V_{sto} + \frac{\bar{w}_{\text{acc}}}{\tau_m}.(t - t^1_{\text{end}}) \quad \text{(D.114)}$$

This behavior leads to a spike at time t^1_{acc} when the condition $V = V_t$ is met:

$$\begin{aligned} V_t &= V_{sto} + \frac{\bar{w}_{\text{acc}}}{\tau_m}.(t^1_{\text{acc}} - t^1_{\text{end}}) && \text{(D.115)}\\ V_t &= V_t.(1-e^{-\Delta T_{cod}/\tau_f}) + \frac{V_t}{T_{\text{cod}}}.(t^1_{\text{acc}} - t^1_{\text{end}}) && \text{(D.116)}\\ t^1_{\text{acc}} &= t^1_{\text{end}} + T_{\text{cod}}.e^{-\Delta T_{cod}/\tau_f}. && \text{(D.117)} \end{aligned}$$

The first output spike is produced by the connection from *last* to *output*, thus:

$$t^1_{\text{out}} = t^1_{\text{last}} + T_{\text{neu}} + 2.T_{\text{syn}} \quad \text{(D.118)}$$

The second output spike is produced by the connection from *acc1* to *output*, thus:

$$\begin{aligned} t^2_{\text{out}} &= t^1_{\text{acc}} + T_{\text{neu}} + T_{\text{syn}} + T_{\min} && \text{(D.119)}\\ &= T_{\text{cod}}.e^{-\Delta T_{cod}/\tau_f} + t^1_{\text{end}} + T_{\text{neu}} + T_{\text{syn}} + T_{\min} && \text{(D.120)}\\ &= T_{\text{cod}}.e^{-\Delta T_{cod}/\tau_f} + t^1_{\text{last}} + T_{\text{neu}} + 2.T_{\text{syn}} + T_{\min} && \text{(D.121)} \end{aligned}$$

This gives us the output ΔT_{out}:

$$\begin{aligned} \Delta T_{out} &= t^2_{\text{out}} - t^1_{\text{out}} && \text{(D.122)}\\ &= T_{\text{cod}}.e^{-\Delta T_{cod}/\tau_f} + T_{\min} && \text{(D.123)} \end{aligned}$$

D.4.3 Multiplier

The Multiplier network (see Fig. 6.14) receives 2 different inputs (a pair of spikes) from each input neurons *input1* (t^1_{in1} and t^2_{in1}) and *input2* (t^1_{in2} and t^2_{in2}) such that $\Delta T_{in1} = t^2_{\text{in1}} - t^1_{\text{in1}}$ and $\Delta T_{in2} = t^2_{\text{in2}} - t^1_{\text{in2}}$ encode its 2 inputs. The first layers of the network, composed of the neurons *input*, *first*, *last* and *acc_log* for each inputs are similar to the Logarithm network. Considering the results from Appendix D.4.1, when the 2 inputs have been fed into the network, we get 2 potentials stored in *acc_log1* and *acc_log2* (respectively V_{sto1} and V_{sto2}):

$$\begin{cases} V_{sto1} = V_t.\frac{\Delta T_{cod1}}{T_{\text{cod}}} \\ V_{sto2} = V_t.\frac{\Delta T_{cod2}}{T_{\text{cod}}} \end{cases} \quad \text{(D.124)}$$

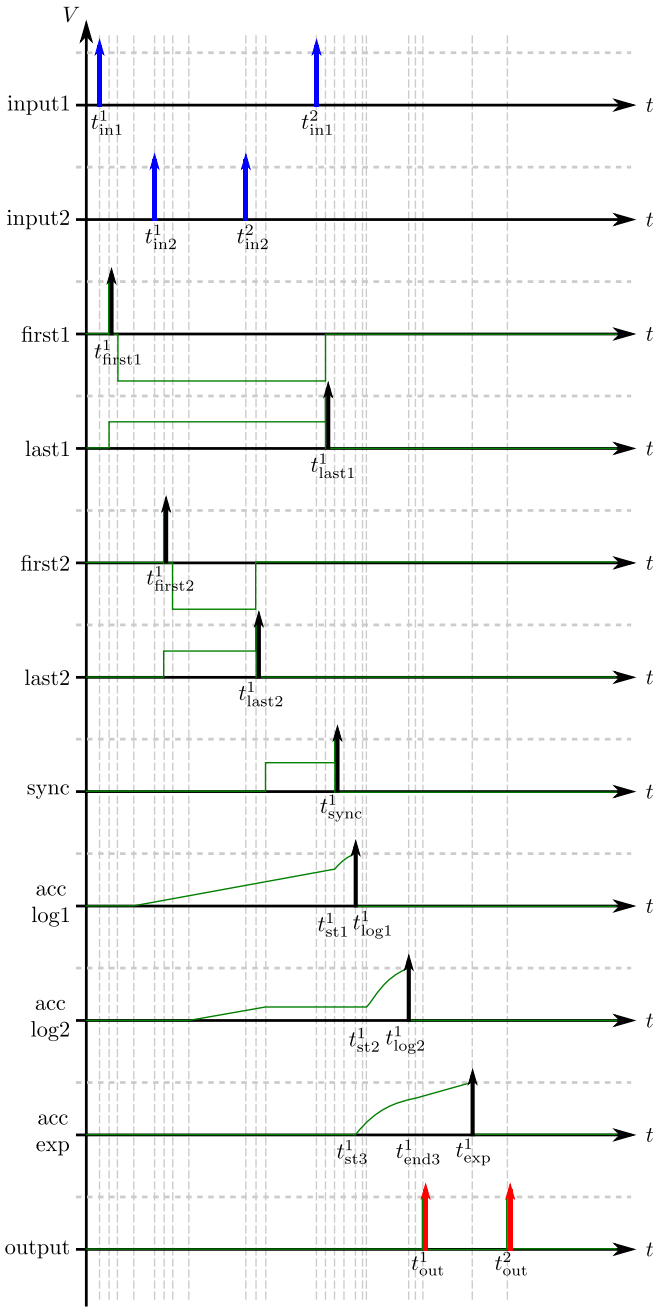


Figure D.10: Multiplier: chronogram of the network for inputs at times t^1_{in1}, t^2_{in1} and t^1_{in2}, t^2_{in2}. (Input spikes are drawn in blue, output spikes in red. Green plots show the membrane potential of interesting neurons).

When the 2 inputs have been fed into the network, spikes from *last1* and *last2* activate the *sync* neuron which spikes at time t^1_{sync}. This triggers the readout of the log of input1 in the *acc_log1*

neuron at time:

$$t^1_{\text{st1}} = t^1_{\text{sync1}} + T_{\text{syn}}. \tag{D.125}$$

Results from Appendix D.4.1 tell us that this neuron will thus spike at time:

$$t^1_{\text{log1}} = -\tau_f . \log\left(\frac{\Delta T_{cod1}}{T_{\text{cod}}}\right) + t^1_{\text{st}}. \tag{D.126}$$

A spike from the *acc_log1* neuron will then trigger the readout of the log value of input2 in the *acc_log2* neuron at time:

$$t^1_{\text{st2}} = t^1_{\text{log1}} + T_{\text{syn}}. \tag{D.127}$$

acc_log2 will then produce a spike at time:

$$t^1_{\text{log2}} = -\tau_f . \log\left(\frac{\Delta T_{cod2}}{T_{\text{cod}}}\right) + t^2_{\text{st}}. \tag{D.128}$$

Which will trigger the first output spike at time:

$$t^1_{\text{out}} = t^1_{\text{log2}} + 2.T_{\text{syn}}. \tag{D.129}$$

At the same time, the *sync* neuron also started the g_f dynamics of neuron *acc_exp* at time:

$$t^1_{\text{st3}} = t^1_{\text{sync}} + 3.T_{\text{syn}}. \tag{D.130}$$

This process is stopped by the spike from neuron *acc_log2* at time:

$$t^1_{\text{end3}} = t^1_{\text{log2}} + T_{\text{syn}} \tag{D.131}$$

Results from Appendix D.4.2 tell us that the potential stored in *acc_exp* at that time is thus:

$$V_{sto3} = V_t.(1 - e^{-(t^1_{\text{end3}} - t^1_{\text{st3}})/\tau_f}) \tag{D.132}$$

and that the integration process started by the second connection from *acc_log2* to *acc_exp* (acting on g_e) will result in a spike at time:

$$t^1_{\text{exp}} = t^1_{\text{end3}} + T_{\text{cod}}.e^{-(t^1_{\text{end3}} - t^1_{\text{st3}})/\tau_f}. \tag{D.133}$$

This will result in the second output spike at time:

$$t^2_{\text{out}} = t^1_{\text{exp}} + T_{\text{syn}} + T_{\text{min}}. \tag{D.134}$$

The output of the network is thus the interspike ΔT_{out} such that:

$$\begin{aligned}
\Delta T_{out} &= t^2_{\text{out}} - t^1_{\text{out}} && \text{(D.135)}\\
&= T_{\text{min}} + t^1_{\text{exp}} - t^1_{\text{log2}} - T_{\text{syn}} && \text{(D.136)}\\
&= T_{\text{min}} + T_{\text{cod}}.e^{-(t^1_{\text{end3}} - t^1_{\text{st3}})/\tau_f} + t^1_{\text{end3}} - T_{\text{syn}} && \text{(D.137)}\\
&= T_{\text{min}} + T_{\text{cod}}.e^{-(t^1_{\text{end3}} - t^1_{\text{st3}})/\tau_f} && \text{(D.138)}
\end{aligned}$$

From previous equations, we get:

$$
\begin{aligned}
t^1_{\text{end3}} - t^1_{\text{st3}} &= t_{\log 2} - t^1_{\text{sync}} - 2.T_{\text{syn}} && \text{(D.139)}\\
&= -\tau_f.\log\left(\frac{\Delta T_{cod2}}{T_{\text{cod}}}\right) + t^2_{\text{st}} - t^1_{\text{sync}} - 2.T_{\text{syn}} && \text{(D.140)}\\
&= -\tau_f.\log\left(\frac{\Delta T_{cod2}}{T_{\text{cod}}}\right) + t^1_{\log 1} - t^1_{\text{sync}} - T_{\text{syn}} && \text{(D.141)}\\
&= -\tau_f.\log\left(\frac{\Delta T_{cod2}}{T_{\text{cod}}}\right) - \tau_f.\log\left(\frac{\Delta T_{cod1}}{T_{\text{cod}}}\right) && \text{(D.142)}\\
&= -\tau_f.\log\left(\frac{\Delta T_{cod1}.\Delta T_{cod2}}{T_{\text{cod}}}\right). && \text{(D.143)}
\end{aligned}
$$

Which then gives us:

$$\Delta T_{out} = T_{\min} + \Delta T_{cod1}.\Delta T_{cod2} \qquad \text{(D.144)}$$

which corresponds to the encoded value of the produce of the value encoded by the 2 inputs.

Bibliography

[1] C. Mead and M.A. Mahowald. A silicon model of early visual processing. *Neural Networks*, 1(1):91–97, 1988.

[2] C Mead. *Analog VLSI and neural systems*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[3] C Mead. Neuromorphic electronic systems. *Proc. of the IEEE*, 78(10):1629–1636, 1990.

[4] J. Lazzaro and C. A. Mead. A silicon model of auditory localization. *Neural Computation*, 1:47–57, 1989.

[5] C. Mead. The silicon retina. *Scientific American*, 1991.

[6] Misha Mahowald. *VLSI analogs of neuronal visual processing: A synthesis of form and function*. PhD thesis, California Institute of Technology, Pasadena, CA, 1992.

[7] Misha Mahowald. *An analog VLSI system for stereoscopic vision*. Kluwer Academic Publishers, Norwell, MA, USA, 1994.

[8] S-C. Liu and T. Delbruck. Neuromorphic sensory systems. *Current Opinion in Neurobiology*, 20(3):288–295, 2010.

[9] Tobi Delbruck. Frame-free dynamic digital vision. In *Proceedings of Intl. Symposium on Secure-Life Electronics, Advanced Electronics for Quality Life and Society*, pages 21–26, Tokyo, Japan, Mar. 2008.

[10] T Delbruck, B Linares-Barranco, E Culurciello, and C Posch. Activity-driven, event-based vision sensors. In *Proc. IEEE International Symposium on Circuits and Systems*, pages 2426–2429, Paris, France, May 2010.

[11] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128x128 120dB 15us Latency Asynchronous Temporal Contrast Vision Sensor. *IEEE J. Solid State Circuits*, 43:566–576, 2008.

[12] C. Posch, D. Matolin, and R. Wohlgenannt. An asynchronous time-based image sensor. In *Proc. IEEE International Symposium on Circuits and Systems*, pages 2130 –2133, Seattle, WA, May 2008.

[13] C. Posch, D. Matolin, and R. Wohlgenannt. A QVGA 143 dB Dynamic Range Frame-Free PWM Image Sensor With Lossless Pixel-Level Video Compression and Time-Domain CDS. *Solid-State Circuits, IEEE Journal of*, 46(1):259 –275, jan. 2011.

[14] Steve Furber, Francesco Galluppi, Steve Temple, and Luis A. Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, 2014.

[15] Xavier Lagorce, Sio-Hoi Ieng, and Ryad Benosman. Event-based features for robotic vision. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 4214–4219. IEEE, 2013.

[16] Xavier Lagorce, Cédric Meyer, Sio-Hoi Ieng, David Filliat, and Ryad Benosman. Asynchronous event-based multikernel algorithm for high-speed visual features tracking. 2014.

[17] Xavier Lagorce, Sio-Hoi Ieng, Xavier Clady, Michael Pfeiffer, and Ryad B Benosman. Spatiotemporal features for asynchronous event-based data. *Frontiers in neuroscience*, 9, 2015.

[18] Xavier Lagorce, Evangelos Stromatias, Francesco Galluppi, Luis A Plana, Shih-Chii Liu, Steve B Furber, and Ryad Benjamin Benosman. Breaking the millisecond barrier on spinnaker: Implementing asynchronous event-based plastic models with microsecond resolution. *Frontiers in Neuroscience*, 9:206, 2015.

[19] Ryad Benosman, Charles Clercq, Xavier Lagorce, Sio-Hoi Ieng, and Chiara Bartolozzi. Event-based visual flow. *IEEE Trans. Neural Netw. Learning Syst.*, 25(2):407–417, 2014.

[20] Francesco Galluppi, Xavier Lagorce, Evangelos Stromatias, Michael Pfeiffer, Luis A Plana, Steve B Furber, and Ryad Benjamin Benosman. A framework for plasticity implementation on the spinnaker neural architecture. *Frontiers in Neuroscience*, 8(429), 2015.

[21] Garrick Orchard, Daniel Matolin, Xavier Lagorce, Ryad Benosman, and Christoph Posch. Accelerated frame-free time-encoded multi-step imaging. In *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, pages 2644–2647. IEEE, 2014.

[22] David Reverter Valeiras, Xavier Lagorce, Xavier Clady, Chiara Bartolozzi, Sio-Hoi Ieng, and Ryad Benosman. An asynchronous neuromorphic event-driven visual part-based shape tracking. 2015.

[23] Garrick Orchard, Xavier Lagorce, Christoph Posch, Ryad Benosman, and Francesco Galluppi. Real-time event-driven spiking neural network object recognition on the spinnaker platform. 2015.

[24] U. Muehlmann, M. Ribo, P. Lang, and A. Pinz. A new high speed cmos camera for real-time tracking applications. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 5195–5200, Barcelona, Spain, Apr. 2004.

[25] H. Oku, N. Ogawa, M. Ishikawa, and K. Hashimoto. Two-dimensional tracking of a motile micro-organism allowing high-resolution observation with various imaging techniques. *Review of Scientific Instruments*, 76(3), 2005.

[26] Patrick Lichtsteiner, Christoph Posch, and Tobias Delbruck. A 128x128 120db 15us latency asynchronous temporal contrast vision sensor. *IEEE Journal of Solid State Circuits*, 43(2):566–576, Feb. 2008.

[27] Juan Antonio Lenero-Bardallo, Teresa Serrano-Gotarredona, and Bernabé Linares-Barranco. A 3.6 us Latency Asynchronous Frame-Free Event-Driven Dynamic-Vision-Sensor. *IEEE J. Solid-State Circuits*, 46(6):1443–1455, June 2011.

[28] Andreas G. Andreou Kwabena A. Boahen. A Contrast Sensitive Silicon Retina with Reciprocal Synapses. *Adv. Neural Inf. Process. Syst.*, pages 764–772, 1991.

[29] A A Stocker. Analog Integrated 2-{D} Optical Flow Sensor. *Analog Integr. Circuits Signal Process.*, 46(2):121–138, 2006.

[30] Kwabena A. Boahen. Point-to-point connectivity between neuromorphic chips using address events. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 47(5):416–434, 2000.

[31] Ankur Handa, Richard A. Newcombe, Adrien Angeli, and Andrew J. Davison. Real-time camera tracking: When is high frame-rate best? In Andrew W. Fitzgibbon, Svetlana Lazebnik, Pietro Perona, Yoichi Sato, and Cordelia Schmid, editors, *ECCV (7)*, volume 7578 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 2012.

[32] T. Komuro, I. Ishii, M. Ishikawa, and A. Yoshida. A digital vision chip specialized for high-speed target tracking. *IEEE Transactions on Electron Devices*, 50:191–199, Jan. 2003.

[33] T.G. Constandinou and C. Toumazou. A micropower centroiding vision processor. *IEEE Journal of Solid-State Circuits*, 41:1430–1443, Jun. 2006.

[34] M. Litzenberger, A.N. Belbachir, N. Donath, G. Gritsch, H. Garn, B. Kohn, C. Posch, and S. Schraml. Estimation of vehicle speed based on asynchronous data from a silicon retina optical sensor. In *IEEE Intelligent Transportation Systems Conference*, pages 653–658, Toronto, Canada, Sep. 2006.

[35] M. Litzenberger, C. Posch, D. Bauer, A.N. Belbachir, P. Schon, B. Kohn, and H. Garn. Embedded vision system for real-time object tracking using an asynchronous transient vision sensor. In *Digital Signal Processing Workshop, 12th - Signal Processing Education Workshop, 4th*, pages 173–178, Teton national Park, WY, Sep. 2006.

[36] T. Delbruck and P. Lichtsteiner. Fast sensory motor control based on event-based hybrid neuromorphic-procedural system. In *IEEE International Symposium on Circuits and Systems*, pages 845–848, New Orleans, LA, May 2007.

[37] http://jaerproject.net/. *Open source jAER software project*.

[38] J. Conradt, M. Cook, R. Berner, P. Lichtsteiner, R.J. Douglas, and T. Delbruck. A pencil balancing robot using a pair of aer dynamic vision sensors. In *IEEE International Symposium on Circuits and Systems*, pages 781–784, Taipei, Taiwan, May 2009.

[39] Z. Ni, C. Pacoret, R. Benosman, S. Ieng, and S. Regnier. Asynchronous event-based high speed vision for microparticle tracking. *Journal of Microscopy*, 245(3):236–244, Mar. 2012.

[40] David Drazen, Patrick Lichtsteiner, Philipp Häfliger, Tobi Delbrück, and Atle Jensen. Toward real-time particle tracking using an event-based dynamic vision sensor. *Experiments in Fluids*, 51(5):1465–1469, Nov. 2011.

[41] Z. Ni, A. Bolopion, J. Agnus, R. Benosman, and S. Regnier. Asynchronous event-based visual shape tracking for stable haptic feedback in microrobotics. *IEEE Transactions on Robotics*, 28:1081–1089, Oct. 2012.

[42] M. Fashing and C. Tomasi. Mean shift is a bound optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:471–474, Mar. 2005.

[43] D. Comaniciu, V. Ramesh, and P. Meer. Kernel-based object tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25:564–577, May 2003.

[44] Huiyu Zhou, Yuan Yuan, and Chunmei Shi. Object tracking using sift features and mean shift. *Comput. Vis. Image Underst.*, 113(3):345–352, March 2009.

[45] K. Krishnamoorthy. *Handbook of Statistical Distributions with Applications*. Chapman and Hall/CRC, Boca Raton, FL, 2006.

[46] D H Hubel and T N Wiesel. Receptive Fields, Binocular Interation and Functional Architectire of the Cat's Cortex. *Journal of Physiology*, 160:106–154, 1962.

[47] M. A. Fischler and R. A. Elschlager. The representation and matching of pictorial structures. *IEEE Transaction on Computers*, 22:67–92, Jan. 1973.

[48] Guy Wallis and Heinrich Bülthoff. Learning to recognize objects. *Trends in Cognitive Science*, 3(1):22–31, 1999.

[49] Bruno A Olshausen and David J Field. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision research*, 37(23):3311–3326, 1997.

[50] Tim Gollisch and Markus Meister. Rapid neural coding in the retina with relative spike latencies. *Science*, 319(5866):1108–1111, 2008.

[51] B. Roska and F. Werblin. Rapid global shifts in natural scenes block spiking in specific ganglion cell types. *Nature Neurosci*, 6:600–608, 2003.

[52] M.J. Berry, D. K. Warland, and M. Meister. The structure and precision of retinal spike trains. *Proceedings of the National Academy of Sciences of the United States of America*, 94:5411–5416, 1997.

[53] U. J. Uzzell and E. J. Chichilnisky. Precision of spike trains in primate retinal ganglion cells. *Journal of Neurophysiology*, 92:780–789, 2004.

[54] G. Wallis and E. Rolls. A model of invariant object recognition in the visual system. *Prog. Neurobiol.*, 51:167–194, 1997.

[55] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980.

[56] M Riesenhuber and T Poggio. Hierarchical models of object recognition in cortex. *Nature Neuroscience*, 2(11):1019–1025, november 1999.

[57] D. Gabor. Theory of communication. *J. IEE*, 93:429–459, 1946.

[58] J. Ilonen, J.-K. Kamarainen, and H. Kalviainen. Fast extraction of multi-resolution gabor features. In *Image Analysis and Processing, 2007. ICIAP 2007. 14th International Conference on*, pages 481 –486, september 2007.

[59] L. Huang, A. Shimizu, and H. Kobatake. Classification-based face detection using Gabor filter features. In *Proc. of IEEE Conf. on Automatic Face and Gesture Recognition*, pages 397 – 402, may 2004.

[60] Thomas Serre, Lior Wolf, Stanley Bileschi, Maximilian Riesenhuber, and Tomaso Poggio. Robust object recognition with cortex-like mechanisms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(3):411–426, 2007.

[61] J. Mutch, U. Knoblich, and T. Poggio. CNS: a GPU-based framework for simulating cortically-organized networks. CBCL 286, MIT Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA, February 2010.

[62] C. Lin and C. Huang. A complex texture classification algorithm based on gabor-type filtering cellular neural networks and self-organized fuzzy inference neural networks. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, volume 4, pages 3942 – 3945, may 2005.

[63] Christoph Posch, Teresa Serrano-Gotarredona, Bernabe Linares-Barranco, and Tobi Delbruck. Retinomorphic event-based vision sensors: Bioinspired cameras with spiking output. *Proceedings of the IEEE*, 102(10):1470–1484, 2014.

[64] David G Lowe. Object recognition from local scale-invariant features. In *Proc. of the IEEE Int. Conf. on Computer Vision (ICCV)*, volume 2, pages 1150–1157, 1999.

[65] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004. 10.1023/B:VISI.0000029664.99615.94.

[66] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, 2008.

[67] B. Schrauwen, D. Verstraeten, and J Van Campenhout. An overview of reservoir computing : theory, applications and implementations. In *Proceedings of the 15th European Sympsosium on Artificial Neural Networks*, pages 471–482, 2007.

[68] H. Jaeger. Tutorial on training recurrent neural networks, covering bppt, rtrl, ekf and the "echo state netwrok" approach. Technical report, German National Research Center for Information Technology, 2002.

[69] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004.

[70] Behrouz Farhang-Boroujeny. *Adaptive filters: theory and applications*. John Wiley & Sons, 2013.

[71] Robert Coultrip, Richard Granger, and Gary Lynch. A cortical model of winner-take-all competition via lateral inhibition. *Neural Networks*, 5(1):47–54, January 1992.

[72] Rodney J Douglas, Misha A Mahowald, and Kevan A C Martin. Hybrid analog-digital architectures for neuromorphic systems. In *in Proc. IEEE World Congress on Computational Intelligence*, pages 1848–1853. IEEE, 1994.

[73] Shih-Chii Liu and Matthias Oster. Feature competition in a spike-based winner-take-all VLSI network. In *Proc. of ISCAS*, pages 3634–3637, 2006.

[74] Matthias Oster, Rodney J. Douglas, and Shih-Chii Liu. Computation with spikes in a winner-take-all network. *Neural Computation*, 21(9):2437–2465, 2009.

[75] Jürgen Schmidhuber. Learning factorial codes by predictability minimization. *Neural Computation*, 4:863–879, 1991.

[76] H. B. Barlow. Unsupervised learning. *Neural Computation*, 1:295–311, 1989.

[77] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.

[78] Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6(2):181–214, 1994.

[79] Seniha Esen Yuksel, Joseph N Wilson, and Paul D Gader. Twenty years of mixture of experts. *IEEE Trans. on Neural Networks and Learning Systems*, 23(8):1177–1193, 2012.

[80] Masahiko Haruno, D Wolpert, and Mitsuo Kawato. MOSAIC model for sensorimotor learning and control. *Neural Computation*, 13(10):2201–2220, 2001.

[81] Kenji Doya, Kazuyuki Samejima, Ken-ichi Katagiri, and Mitsuo Kawato. Multiple model-based reinforcement learning. *Neural Computation*, 14(6):1347–1369, 2002.

[82] Eiji Uchibe and Kenji Doya. Competitive-cooperative-concurrent reinforcement learning with importance sampling. In *Proc. of International Conference on Simulation of Adaptive Behavior: From Animals and Animats*, pages 287–296, 2004.

[83] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society B*, pages 1–38, 1977.

[84] Bernhard Nessler, Michael Pfeiffer, and Wolfgang Maass. STDP enables spiking neurons to detect hidden causes of their inputs. In *Proc. of NIPS*, pages 1357–1365, 2009.

[85] Bernhard Nessler, Michael Pfeiffer, Lars Buesing, and Wolfgang Maass. Bayesian computation emerges in generic cortical microcircuits through spike-timing-dependent plasticity. *PLoS Computational Biology*, 9(4):e1003037, 2013.

[86] Clément Farabet, Rafael Paz, Jose Pérez-Carrasco, Carlos Zamarreño-Ramos, Alejandro Linares-Barranco, Yann LeCun, Eugenio Culurciello, Teresa Serrano-Gotarredona, and Bernabe Linares-Barranco. Comparison between frame-constrained fix-pixel-value and frame-free spiking-dynamic-pixel convnets for visual processing. *Frontiers in Neuroscience*, 6, 2012.

[87] Luis A Camuñas-Mesa, Teresa Serrano-Gotarredona, Sio H Ieng, Ryad B Benosman, and Bernabe Linares-Barranco. On the use of orientation filters for 3d reconstruction in event-driven stereo vision. *Frontiers in Neuromorphic Engineering*, 8, 2014.

[88] Peter O'Connor, Daniel Neil, Shih-Chii Liu, Tobi Delbruck, and Michael Pfeiffer. Real-time classification and sensor fusion with a spiking deep belief network. *Frontiers in Neuromorphic Engineering*, 7, 2013.

[89] Jonathan C Tapson, Greg Kevin Cohen, Saeed Afshar, Klaus M Stiefel, Yossi Buskila, Tara Julia Hamilton, and André van Schaik. Synthesis of neural networks for spatio-temporal spike pattern recognition and processing. *Frontiers in Neuroscience*, 7(153), 2013.

[90] Sadique Sheik, Michael Pfeiffer, Fabio Stefanini, and Giacomo Indiveri. Spatio-temporal spike pattern classification in neuromorphic systems. In *Biomimetic and Biohybrid Systems*, pages 262–273. Springer, 2013.

[91] David Kappel, Bernhard Nessler, and Wolfgang Maass. STDP installs in winner-take-all circuits an online approximation to hidden markov model learning. *PLoS Computational Biology*, 10(3):e1003511, 2014.

[92] Dane Sterling Corneil, Emre Neftci, Giacomo Indiveri, and Michael Pfeiffer. Learning, inference, and replay of hidden state sequences in recurrent spiking neural networks. In *COSYNE 2014*, pages 1–2, 2014.

[93] Thomas Serre, Maximilian Riesenhuber, Jennifer Louie, and Tomaso Poggio. On the role of object-specific features for real world object recognition in biological vision. In *Proc. of BMCV*, pages 387–397, 2002.

[94] Jun Haeng Lee, Tobi Delbruck, Michael Pfeiffer, Paul KJ Park, C-W Shin, H Ryu, and Byung Chang Kang. Real-time gesture interface based on event-driven processing from stereo silicon retinas. *IEEE Transactions on Neural Networks and Learning Systems*, 25:2250 – 2263, 2014.

[95] Elias Mueggler, Basil Huber, and Davide Scaramuzza. Event-based, 6-DOF pose tracking for high-speed maneuvers. In *Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 2761–2768, 2014.

[96] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.

[97] Wolfgang Maass and Henry Markram. On the computational power of circuits of spiking neurons. *Journal of computer and system sciences*, 69(4):593–616, 2004.

[98] Lars Büsing, Benjamin Schrauwen, and Robert Legenstein. Connectivity, dynamics, and memory in reservoir computing with binary and analog neurons. *Neural computation*, 22(5):1272–1311, 2010.

[99] G Indiveri, E Chicca, and R Douglas. A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity. *IEEE Transactions on Neural Networks*, 17:211–221, 2006.

[100] M Rahimi Azghadi, Nicolangelo Iannella, Said F Al-Sarawi, Giacomo Indiveri, and Derek Abbott. Spike-based synaptic plasticity in silicon: Design, implementation, application, and challenges. *Proceedings of the IEEE*, 102(5):717–737, 2014.

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

[102] Vasudev Bhaskaran and Konstantinos Konstantinides. *Image and Video Compression Standards: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA, 2nd edition, 1997.

[103] Erik Reinhard, Wolfgang Heidrich, Paul Debevec, Sumanta Pattanaik, Greg Ward, and Karol Myszkowski. *High dynamic range imaging: acquisition, display, and image-based lighting*. Morgan Kaufmann, 2010.

[104] Tim Gollisch and Markus Meister. Eye smarter than scientists believed: Neural computations in circuits of the retina. *Neuron*, 65(2):150 – 164, 2010.

[105] Himanshu Akolkar, Cedric Meyer, Zavier Clady, Olivier Marre, Chiara Bartolozzi, Stefano Panzeri, and Ryad Benosman. What can neuromorphic event-driven precise timing add to spike-based pattern recognition? *Neural computation*, 2015.

[106] Rafael Serrano-Gotarredona, Matthias Oster, Patrick Lichtsteiner, Alejandro Linares-Barranco, Rafael Paz-Vicente, Francisco Gomez-Rodriguez, Luis Camunas-Mesa, Raphael Berner, Manuel Rivas-Perez, Tobi Delbruck, Shih-Chii Liu, Rodney Douglas, Philipp Hafliger, Gabriel Jimenez-Moreno, Anton Civit Ballcels, Teresa Serrano-Gotarredona, Antonio J Acosta-Jimenez, and Bernabé Linares-Barranco. CAVIAR: a 45k neuron, 5M synapse, 12G connects/s AER hardware sensory-processing- learning-actuating system for high-speed visual object recognition and tracking. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 20(9):1417–38, September 2009.

[107] Fopefolu Folowosele, R. Jacob Vogelstein, and Ralph Etienne-Cummings. Towards a Cortical Prosthesis: Implementing A Spike-Based HMAX Model of Visual Object Recognition in Silico. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(4):516–525, December 2011.

[108] José Antonio Pérez-Carrasco, Bo Zhao, Carmen Serrano, Begoña Acha, Teresa Serrano-Gotarredona, Shouchun Chen, and Bernabé Linares-Barranco. 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):2706–19, November 2013.

[109] Garrick Orchard, Cedric Meyer, Ralph Etienne-Cummings, Christoph Posch, Nitish Thakor, and Ryad Benosman. Hfirst: A temporal approach to object recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PP, 2015.

[110] Peter O'Connor, Daniel Neil, Shih-Chii Liu, Tobi Delbruck, and Michael Pfeiffer. Real-time classification and sensor fusion with a spiking deep belief network. *Frontiers in neuroscience*, 7:178, January 2013.

[111] Shoushun Chen, Polina Akselrod, Bo Zhao, Jose Antonio Perez Carrasco, Bernabe Linares-Barranco, and Eugenio Culurciello. Efficient feedforward categorization of objects and human postures with address-event image sensors. *IEEE transactions on pattern analysis and machine intelligence*, 34(2):302–14, February 2012.

[112] R. Ghosh, A. Mishra, G. Orchard, and V. Thakor. Real-time object recognition and orientation estimation using an event-based camera and cnn. *IEEE Biomedical Circuits and Systems*, 2014.

[113] Teresa Serrano-Gotarredona and Bernabé Linares-Barranco. A 128 x 128 1.5% contrast sensitivity 0.9% fpn 3 μs latency 4 mw asynchronous frame-free dynamic vision sensor using transimpedance preamplifiers. *Solid-State Circuits, IEEE Journal of*, 48(3):827–838, 2013.

[114] Dana Ballard and Janneke Jehee. Dynamic coding of signed quantities in cortical feedback circuits. *Frontiers in Psychology*, 3(254), 2012.

[115] Jaeger H. The "echo state" approach to analysing and training recurrent neural networks. *GMD Report 148, GMD - German National Research Institute for Computer Science*, 2001.

[116] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for improved unconstrained handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5), 2009.

[117] A. Bhattacharyya. On a measure of divergence between two statistical populations defined by probability distributions. *Bull. Calcutta Math. Soc*, 35:99–109, 1943.

[118] J. A. Perez-Carrasco, B. Zhao, C. Serrano, B. Acha, T. Serrano-Gotarredona, S. Chen, and B. Linares-Barranco. Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward convnets. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(11):2706–2719, November 2013.

[119] Timothée Masquelier and Simon J Thorpe. Unsupervised learning of visual features through spike timing dependent plasticity. *PLoS computational biology*, 3(2):e31, 2007.

[120] SB Furber, F Galluppi, S Temple, and A Plana. The SpiNNaker Project. *Proceedings of the IEEE*, 102(5):652–665, May 2014.

[121] Donald O Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Wiley-Interscience, New York, new editio edition, June 1949.

[122] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–2558, April 1982.

[123] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, 15(3):267–273, 1982.

[124] E L Bienenstock, L N Cooper, and P W Munro. Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 2(1):32–48, 1982.

[125] H Markram, J Lbke, M Frotscher, and B Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. *Science*, 275(5297):213–215, January 1997.

[126] GQ Bi and MM Poo. Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of Neuroscience*, 18(24):10464–10472, December 1998.

[127] Claudia Clopath, Lars Busing, Eleni Vasilaki, and Wulfram Gerstner. Connectivity reflects coding: a model of voltage-based STDP with homeostasis. *Nature Neuroscience*, 13(3):344–352, March 2010.

[128] S Song and L F Abbott. Cortical development and remapping through spike timing-dependent plasticity. *Neuron*, 32(2):339–350, October 2001.

[129] W Gerstner, R Kempter, J L van Hemmen, and H Wagner. A Neuronal Learning Rule for Sub-millisecond Temporal Coding. *Nature*, 383(6595):76–78, 1996.

[130] Rudy Guyonneau, Rufin Van Rullen, and Simon J Thorpe. Neurons tune to the earliest spikes through STDP. *Neural Computation*, 17(4):859–879, 2005.

[131] S Song, K D Miller, and L F Abbott. Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neuroscience*, 3(9):919–926, September 2000.

[132] R Kempter, W Gerstner, and J.˜L. van Hemmen. Intrinsic stabilization of output rates by spike-based Hebbian learning. *Neural Computation*, 13:2709–2741, 2001.

[133] EM Izhikevich. Solving the distal reward problem through linkage of STDP and dopamine signaling. *Cereb Cortex*, 17(10):2443–2452, October 2007.

[134] Robert Legenstein, Dejan Pecevski, and Wolfgang Maass. A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback. *PLoS Computational Biology*, 4(10):e1000180, 2008.

[135] Johannes Friedrich, Robert Urbanczik, and Walter Senn. Spatio-temporal credit assignment in neuronal population learning. *PLoS Computational Biology*, 7(6):e1002092, 2011.

[136] W Potjans, M Diesmann, and A Morrison. An Imperfect Dopaminergic Error Signal Can Drive Temporal-Difference Learning. *PLoS Computational Biology*, 7(5):20, 2011.

[137] L F Abbott and S B Nelson. Synaptic plasticity: taming the beast. *Nature Neuroscience*, 3:1178–1183, 2000.

[138] Dimitri M Kullmann, Alexandre W Moreau, Yamina Bakiri, and Elizabeth Nicholson. Plasticity of inhibition. *Neuron*, 75(6):951–962, 2012.

[139] Verena Pawlak, Jeffery R Wickens, Alfredo Kirkwood, and Jason N D Kerr. Timing is not everything: neuromodulation opens the STDP gate. *Frontiers in Synaptic Neuroscience*, 2, 2010.

[140] Stijn Cassenaer and Gilles Laurent. Conditional modulation of spike-timing-dependent plasticity for olfactory learning. *Nature*, 482(7383):47–52, 2012.

[141] P J Sjöström, G G Turrigiano, and S B Nelson. Rate, timing, and cooperativity jointly determine cortical synaptic plasticity. *Neuron*, 32:1149–1164, 2001.

[142] Joseph M Brader, Walter Senn, and Stefano Fusi. Learning real-world stimuli in a neural network with spike-driven synaptic dynamics. *Neural computation*, 19(11):2881–912, November 2007.

[143] Michael Beyeler, Nikil D Dutt, and Jeffrey L Krichmar. Categorization and decision-making in a neurobiologically plausible spiking network using a STDP-like learning rule. *Neural Networks*, 48:109–124, 2013.

[144] Sadique Sheik, Martin Coath, Giacomo Indiveri, Susan L. Denham, Thomas Wennekers, and Elisabetta Chicca. Emergent Auditory Feature Tuning in a Real-Time Neuromorphic VLSI System. *Frontiers in Neuroscience*, 6, 2012.

[145] Bernhard Nessler, Michael Pfeiffer, Lars Buesing, and Wolfgang Maass. Bayesian computation emerges in generic cortical microcircuits through spike-timing-dependent plasticity. *PLoS computational biology*, 9(4):e1003037, 2013.

[146] David Kappel, Bernhard Nessler, and Wolfgang Maass. STDP Installs in Winner-Take-All Circuits an Online Approximation to Hidden Markov Model Learning. *PLoS Computational Biology*, 10(3):e1003511, 2014.

[147] C Savin, P Joshi, and J Triesch. Independent Component Analysis in Spiking Neurons. *PLoS Computational Biology*, 6(4):e1000757, 2010.

[148] Emre Neftci, Srinjoy Das, Bruno Pedroni, Kenneth Kreutz-Delgado, and Gert Cauwenberghs. Event-driven contrastive divergence for spiking neuromorphic systems. *Frontiers in Neuroscience*, 7, 2014.

[149] A Morrison, C Mehring, and T Geisel. Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Computation*, 17(8):1776–1801, 2005.

[150] R Brette, M Rudolph, T Carnevale, M Hines, D Beeman, JM Bower, M Diesmann, A Morrison, PH Goodman, FC Harris, M Zirpe, T Natschlger, D Pecevski, B Ermentrout, M Djurfeldt, A Lansner, O Rochel, T Vieville, E Muller, AP Davison, S Boustani, and A Destexhe. Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of Computational Neuroscience*, 23(3):349–398, December 2007.

[151] H. Plesser, J. Eppler, A. Morrison, M. Diesmann, and M.O. Gewaltig. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. *Euro-Par 2007 parallel processing*, pages 672–681, 2007.

[152] TM Wong, R Preissl, P Datta, M Flickner, R Singh, SK Esser, E McQuinn, R Appuswamy, WP Risk, HD Simon, and DS Modha. 10ˆ14. Technical report, IBM, 2013.

[153] AK Fidjeland and M Shanahan. Accelerated simulation of spiking neural networks using GPUs. In *Neural Networks, International Joint Conference on*, pages 1–8, 2010.

[154] Daniel Neil and Shih-Chii Liu. Minitaur, an Event-Driven FPGA-Based Spiking Network Accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–1, 2014.

[155] Giacomo Indiveri, Bernabé Linares-Barranco, Tara Julia Hamilton, André Van Schaik, Ralph Etienne-Cummings, Tobi Delbruck, Shih-Chii SC Liu, Piotr Dudek, Philipp Häfliger, Sylvie Renaud, Johannes Schemmel, Gert Cauwenberghs, John Arthur, Kai Hynna, Fopefolu Folowosele, Sylvain Saighi, Teresa Serrano-Gotarredona, Jayawan Wijekoon, Yingxue Wang, Kwabena Boahen, and Boahen. Neuromorphic Silicon Neuron Circuits. *Front. Neurosci.*, 5(May):23, 2011.

[156] SB Furber, S Temple, and AD Brown. High-Performance Computing for Systems of Spiking Neurons. *The AISB06 workshop on GC5: Architecture of Brain and Mind*, 2006.

[157] A Morrison, M Diesmann, and W Gerstner. Phenomenological models of synaptic plasticity based on spike timing. *Biological cybernetics*, 98(6):459–478, 2008.

[158] J. Schemmel, D. Bruderle, K. Meier, and B. Ostendorf. Modeling Synaptic Plasticity within Networks of Highly Accelerated I&F Neurons. *2007 IEEE International Symposium on Circuits and Systems*, 2007.

[159] Thomas Pfeil, Tobias C. Potjans, Sven Schrader, Wiebke Potjans, Johannes Schemmel, Markus Diesmann, and Karlheinz Meier. Is a 4-Bit Synaptic Weight Resolution Enough? – Constraints on Enabling Spike-Timing Dependent Plasticity in Neuromorphic Hardware, 2012.

[160] R Jacob Vogelstein, Francesco Tenore, Ralf Philipp, Miriam S Adlerstein, David H Goldberg, and Gert Cauwenberghs. Spike timing-dependent plasticity in the address domain. In *Advances in Neural Information Processing Systems*, pages 1147–1154, 2002.

[161] M. Giulioni, P. Camilleri, V. Dante, D. Badoni, G. Indiveri, J. Braun, and P. Del Giudice. A VLSI network of spiking neurons with plastic fully configurable learning synapses. *2008 15th IEEE International Conference on Electronics, Circuits and Systems*, 2008.

[162] H Mostafa, F Corradi, F Stefanini, and G Indiveri. A hybrid analog/digital Spike-Timing Dependent Plasticity learning circuit for neuromorphic VLSI multi-neuron architectures. In *International Symposium on Circuits and Systems (ISCAS) 2014*, pages 854–857, 2014.

[163] X Jin, F Galluppi, C Patterson, A Rast, S Davies, S Temple, and SB Furber. Algorithm and Software for Simulation of Spiking Neural Networks on the Multi-Chip SpiNNaker System. In *Neural Networks, 2010. IJCNN 2010. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 1–8, September 2010.

[164] Mostafa Rahimi Azghadi, Nicolangelo Iannella, Said F. Al-Sarawi, Giacomo Indiveri, and Derek Abbott. Spike-Based Synaptic Plasticity in Silicon: Design, Implementation, Application, and Challenges. *Proceedings of the IEEE*, 102(5):717–737, May 2014.

[165] Carlos Zamarreño Ramos, Luis A Camuñas Mesa, Jose A Pérez-Carrasco, Timothée Masquelier, Teresa Serrano-Gotarredona, and Bernabé Linares-Barranco. On spike-timing-dependent-plasticity, memristive devices, and building a self-learning visual cortex. *Frontiers in neuroscience*, 5:26, 2011.

[166] Christian Mayr, Paul Stärke, Johannes Partzsch, Rene Schueffny, Love Cederstroem, Yao Shuai, Nan Du, and Heidemarie Schmidt. Waveform Driven Plasticity in BiFeO3 Memristive Devices: Model and Implementation. In *NIPS*, pages 1709–1717, 2012.

[167] Giacomo Indiveri, Bernabé Linares-Barranco, Robert Legenstein, George Deligeorgis, and Themistoklis Prodromakis. Integration of nanoscale memristor synapses in neuromorphic computing architectures. *IOP Nanotechnology*, 24(38):384010, 2013.

[168] JM Nageswaran, N Dutt, JL Krichmar, and A Nicolau. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks*, 22(5–6), 2007.

[169] EM Izhikevich. Polychronization: Computation with Spikes. *Neural Computation*, 18(2):245–282, February 2006.

[170] SB Furber and S Temple. Neural Systems Engineering. *Computational Intelligence: A Compendium*, 4(13):763–796, April 2008.

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

[172] E Painkras, LA Plana, J Garside, S Temple, F Galluppi, C Patterson, DR Lester, AD Brown, and SB Furber. SpiNNaker : A 1W 18-core System-on-Chip for Massively-Parallel Neural Net Simulation. *The IEEE Journal of Solid State Circuits*, VV(8):1–13, August 2013.

[173] T Sharp, LA Plana, F Galluppi, and SB Furber. Event-Driven Simulation of Arbitrary Spiking Neural Networks on SpiNNaker. In *The International Conference on Neural Information Processing (ICONIP)*, volume 2011, pages 424–430. Springer, 2011.

[174] F Galluppi, S Davies, AD Rast, T Sharp, LA Plana, and S.B. Furber. A Hierarchical Configuration System for a Massively Parallel Neural Hardware Platform. In ACM, editor, *CF '12 Proceedings of the 9th conference on Computing Frontiers*, pages 183–192, Cagliari, 2012.

[175] AP Andrew P Davison, Daniel Brüderle, JM Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: A Common Interface for Neuronal Network Simulators. *Front Neuroinformatics*, 2(0):11, 2008.

[176] Terrence C Stewart, Bryan Tripp, and Chris Eliasmith. Python scripting in the Nengo simulator. *Frontiers in Neuroinformatics*, 3(0), 2009.

[177] X Jin, A Rast, F Galluppi, M Khan, and SB Furber. Implementing Learning on the SpiNNaker Universal Neural Chip Multiprocessor. In Chi S Leung, Minho Lee, and Jonathan H Chan, editors, *Neural Information Processing*, volume 5863, chapter 48, pages 425–432. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[178] X. Jin, A Rast, F Galluppi, S Davies, and SB Furber. Implementing Spike-Timing-Dependent Plasticity on SpiNNaker Neuromorphic Hardware. In *Neural Networks, 2010. IJCNN 2010. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 1–8. IEEE, 2010.

[179] S. Davies, F. Galluppi, A.D. Rast, and SB Furber. A forecast-based STDP rule suitable for neuromorphic implementation. *Neural Networks*, 32:3–14, 2012.

[180] M R Mehta, M C Quirk, and M A Wilson. Experience-dependent asymmetric shape of hippocampal receptive fields. *Neuron*, 25(3):707–715, 2000.

[181] Harel Shouval, Nathan Intrator, and Leon N. Cooper. BCM network develops orientation selectivity and ocular dominance in natural scene environment. *Vision Research*, 37(23):3339–3342, 1997.

[182] H K Hartline, Henry G Wagner, and Floyd Ratliff. Inhibition in the Eye of the Limulus. *The Journal of General Physiology*, 39(5):651–673, 1956.

[183] Jeyadarshan Jeyabalaratnam, Vishal Bharmauria, Lyes Bachatene, Sarah Cattan, Annie Angers, and Stéphane Molotchnikoff. Adaptation Shifts Preferred Orientation of Tuning Curve in the Mouse Visual Cortex. *PLoS ONE*, 8(5):e64294, 2013.

[184] Bartlett D Moore and Ralph D Freeman. Development of orientation tuning in simple cells of primary visual cortex. *Journal of Neurophysiology*, 107(9):2506–2516, 2012.

[185] G Indiveri and S Fusi. Spike-based learning in VLSI networks of integrate-and-fire neurons. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 3371–3374, May 2007.

[186] S. Mitra, S. Fusi, and G. Indiveri. Real-Time Classification of Complex Patterns Using Spike-Based Learning in Neuromorphic VLSI. *IEEE Transactions on Biomedical Circuits and Systems*, 3(1), 2009.

[187] Massimilian Giulioni, Mario Pannunzi, Davide Badoni, Vittorio Dante, and Paolo Del Giudice. Classification of correlated patterns with a configurable analog VLSI neural network of spiking neurons and self-regulating plastic synapses. *Neural computation*, 21(11):3106–3129, 2009.

[188] Martin Coath, Sadique Sheik, Elisabetta Chicca, Giacomo Indiveri, Susan L Denham, and Thomas Wennekers. A robust sound perception model suitable for neuromorphic implementation. *Frontiers in neuroscience*, 7(January):278, 2013.

[189] A van Schaik and SC Liu. AER EAR: A matched silicon cochlea pair with address event representation interface. In *IEEE International Symposium on Circuits and Systems ISCAS*, pages 4213–4216, 2005.

[190] Peter U Diehl and Matthew Cook. Efficient Implementation of STDP Rules on SpiNNaker Neuromorphic Hardware. In *International Conference on Neural Networks (IJCNN) 2014*, pages 4288–4295, 2014.

[191] T Sharp and S Furber. Correctness and performance of the SpiNNaker architecture. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–8, August 2013.

[192] E Stromatias, F Galluppi, C Patterson, and SB Furber. Power analysis of large-scale, real-time neural networks on SpiNNaker. In *The International Joint Conference on Neural Networks - IJCNN 213*, pages 1570–1577, 2013.

[193] Wulfram Gerstner, Henning Sprekeler, and Gustavo Deco. Theory and simulation in neuroscience. *Science*, 338(6103):60–65, 2012.

[194] Q V Le, R Monga, M Devin, G Corrado, K Chen, M Ranzato, J Dean, and A Y Ng. Building high-level features using large scale unsupervised learning. In *Proceedings of the 29th International Conference on Machine Learning (ICML)*, pages 1–11, 2012.

[195] AK Fidjeland, EB. Roesch, M Shanahan, and W Luk. NeMo: A Platform for Neural Modelling of Spiking Neurons Using GPUs. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 137–144. IEEE, July 2009.

[196] F Galluppi, C Denk, M C Meiner, T Stewart, L A Plana, C Eliasmith, S Furber, and J Conradt. Event-based neural computing on an autonomous mobile platform. In *Proc. of IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–6, 2014.

[197] Romain Brette and Dan FM Goodman. Simulating spiking neural networks on gpu. *Network: Computation in Neural Systems*, 23(4):167–182, 2012.

[198] Chiara Bartolozzi, Olga Nikolayeva, and Giacomo Indiveri. Implementing homeostatic plasticity in VLSI networks of spiking neurons. In *Proceedings of the 15th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2008*, pages 682–685, 2008.

[199] Runchun Wang, Gregory Cohen, Klaus M Stiefel, Tara Julia Hamilton, Jonathan Tapson, and André van Schaik. An FPGA Implementation of a Polychronous Spiking Neural Network with Delay Adaptation. *Frontiers in neuroscience*, 7(February):14, 2013.

[200] Andreea Lazar, Gordon Pipa, and Jochen Triesch. {SORN}: a self-organizing recurrent neural network. *Frontiers in computational neuroscience*, 3:23, 2009.

[201] Florence I Kleberg, Tomoki Fukai, and Matthieu Gilson. Excitatory and inhibitory STDP jointly tune feedforward neural circuits to selectively propagate correlated spiking activity. *Frontiers in Computational Neuroscience*, 8, 2014.

[202] Jonathan Binas, Ueli Rutishauser, Giacomo Indiveri, and Michael Pfeiffer. Learning and Stabilization of Winner-Take-All Dynamics Through Interacting Excitatory and Inhibitory Plasticity. *Frontiers in Computational Neuroscience*, 8(68), 2014.

[203] Rogier Min, Mirko Santello, and Thomas Nevian. The computational power of astrocyte mediated synaptic plasticity, 2012.

[204] IV Tetko and AE Villa. A pattern grouping algorithm for analysis of spatiotemporal patterns in neuronal spike trains. 1. Detection of repeated patterns. *Journal of Neuroscience Methods*, 105(1):1–14, 2001.

[205] EY Chang, KF Morris, R Shannon, and BG Lindsey. Repeated sequences of interspike intervals in baroresponsive respiratory related neuronal assemblies of the cat brain stem. *Journal of Neurophysiology*, 84(3):1136–1148, September 2000.

[206] BQ Mao, F Hamzei-Sichani, D Aronov, RC Froemke, and R Yuste. Dynamics of spontaneous activity in neocortical slices. *Neuron*, 32(5):883–898, December 2001.

[207] Wyeth Bair and Christof Koch. Temporal precision of spike trains in extrastriate cortex of the behaving macaque monkey. *Neural computation*, 1202, 1996.

[208] M J Berry, D K Warland, and M Meister. The structure and precision of retinal spike trains. *Proceedings of the National Academy of Sciences of the United States of America*, 94(10):5411–5416, 1997.

[209] Tim Gollisch and Markus Meister. Rapid neural coding in the retina with relative spike latencies. *Science (New York, N.Y.)*, 319(5866):1108–1111, 2008.

[210] W. Softky. Sub-millisecond coincidence detection in active dendritic trees. *Neuroscience*, 58(1):13–41, 1994.

[211] Eric I Knudsen. Instructed learning in the auditory localization pathway of the barn owl. *Nature*, 417(6886):322–328, 2002.

[212] Yoram Gutfreund, Weimin Zheng, and Eric I Knudsen. Gated visual input to the central auditory system. *Science (New York, N.Y.)*, 297(5586):1556–1559, 2002.

[213] Stephen Brink, Stephen Nease, Paul Hasler, Shubha Ramakrishnan, Richard Wunderlich, Arindam Basu, and Brian Degnan. A learning-enabled neuron array IC based upon transistor channel models of biological phenomena. *IEEE Transactions on Biomedical Circuits and Systems*, 7(1):71–81, 2013.

[214] B V Benjamin, P Gao, E McQuinn, S Choudhary, A R Chandrasekaran, J.-M. Bussat, R Alvarez-Icaza, J V Arthur, P A Merolla, and K Boahen. Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations. *Proceedings of the IEEE*, PP(99):1–18, 2014.

[215] Steve Furber and Steve Temple. Neural systems engineering. *Journal of the Royal Society interface*, 4(13):193–206, 2007.

[216] P. a. Merolla, J. V. Arthur, R. Alvarez-Icaza, a. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, August 2014.

[217] Jian Wu and Steve Furber. A multicast routing scheme for a universal spiking neural network architecture. *Comput. J.*, 53(3):280–288, March 2010.

[218] Steve B. Furber, David R. Lester, Luis A. Plana, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown. Overview of the spinnaker system architecture. *IEEE Trans. Comput.*, 62(12):2454–2467, December 2013.

[219] E. Stromatias, F. Galluppi, C. Patterson, and S. Furber. Power analysis of large-scale, real-time neural networks on spinnaker. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–8, Aug 2013.

[220] Andrew P. Davison, Daniel Brüderle, Jochen M. Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2:11, 2009.

[221] Francesco Galluppi, Sergio Davies, Alexander Rast, Thomas Sharp, Luis A. Plana, and Steve Furber. A hierachical configuration system for a massively parallel neural hardware platform. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 183–192, New York, NY, USA, 2012. ACM.

[222] jAER. jAER Open Source Project. `http://jaerproject.org`, 2007.

[223] C E Carr and M Konishi. Axonal delay lines for time measurement in the owl's brainstem. *Proceedings of the National Academy of Sciences*, 85(21):8311–8315, 1988.

[224] S-C. Liu, A. van Schaik, B.A. Minch, and T. Delbruck. Asynchronous binaural spatial audition sensor with 2 x 64 x 4 channel output. *IEEE Transactions on Biomedical Circuits and Systems*, 8(4):453–464, Aug 2014.

[225] F Galluppi, K Brohan, S Davidson, T Serrano-Gotarredona, JA Carrasco, B Linares-Barranco, and Steve B Furber. A real-time, event-driven neuromorphic system for goal-directed attentional selection. In *Neural Information Processing*, pages 226–233. Springer, 2012.

[226] G Orchard, X Lagorce, C Posch, R Benosman, and F Galluppi. Real-time Event-driven Spiking Neural Network Object Recognition on the SpiNNaker Platform. In *Submitted to the International Conference of Circuit and Systems (ISCAS) 2015*, pages –, 2015.

[227] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. volume 30, pages 1503–1506, 2005.

[228] P.A. Merolla, J.V. Arthur, R. Alvarez-Icaza, A.S. Cassidy, J. Sawada, F. Akopyan, B.L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S.K. Esser, R. Appuswamy, B. Taba, A. Amir, M.D. Flickner, W.P. Risk, R. Manohar, and D.S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, pages 668–673, August 2014.

[229] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V. Arthur, Paul Merolla, and Kwabena Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, pages 699–716, 2014.

[230] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 431 –438, june 2008.

[231] Chris Eliasmith, Terrence C. Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Yichuan Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *Science*, 338:1202–1205, 2012.

[232] G. Indiveri. A current-mode hysteretic winner-take-all network, with excitatory and inhibitory coupling. *Analog Integrated Circuits and Signal Processing*, 28(3):279–291, September 2001.

[233] Carlos Zamarreño-Ramos, Alejandro Linares-Barranco, Teresa Serrano-Gotarredona, and Bernabé Linares-Barranco. Multicasting mesh aer: A scalable assembly approach for reconfigurable neuromorphic structured aer systems. application to convnets. *IEEE Trans. Biomed. Circuits and Systems*, 7(1):82–102, 2013.

[234] Jonathan Tapson and André van Schaik. Learning the pseudoinverse solution to network weights. *CoRR*, abs/1207.3368, 2012.

[235] Guang-Bin Huang and Lei Chen. Enhanced random search based incremental extreme learning machine. *Neurocomput.*, 71(16-18):3460–3468, October 2008.

[236] Ali Rahimi and Benjamin Recht. Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1313–1320. Curran Associates, Inc., 2009.

[237] Andrew Saxe, Pang Wei Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, and Andrew Ng. On random weights and unsupervised feature learning. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 1089–1096, New York, NY, USA, June 2011. ACM.

[238] M. N. Shadlen and W.T. Newsome. The variable discharge of cortical neurons: Implications for connectivity, computation and information coding. volume 18, pages 3870–3896, 1998.

[239] M.E. Mazurek and M.N. Shadlen. Limits to the temporal fidelity of cortical spike rate signals. volume 5, pages 463–471, 2002.

[240] V. Litvak, H. Sompolinsky, I. Segev, and M. Abeles. On the transmission of rate code in long feed-forward networks with excitatory-inhibitory balance. volume 23, pages 3006–3015, 2003.

[241] Terrence J. Sejnowski. Time for a new neural code? *Nature*, 376(6535):21–22, 1995.

[242] Z. F. Mainen and T. J. Sejnowski. Reliability of spike timing in neocortical neurons. volume 268, pages 1503–1506, 1995.

[243] B.G. Lindsey, K.F. Morris, R. Shannon, and G.L. Gerstein. Repeated patterns of distributed synchrony in neuronal assemblies. volume 78, page 1714–1719, 1997.

[244] Y. Prut, E. Vaadia, H. Bergman, I. Haalman, H Slovin, and M. Abeles. Spatiotemporal structure of cortical activity: Properties and behavioral relevance. volume 79, page 2857–2874, 1998.

[245] A. E.and Tetko Villa, I. V., B. Hyland, and A. Najem. Spatiotemporal activity patterns of rat cortical neurons predict responses in a conditioned task. volume 96, page 1106–1111, 1999.

[246] E. Y. Chang, K. F. Morris, R. Shannon, and B. G. Lindsey. Repeated sequences of interspike intervals in baroresponsive respiratory related neuronal assemblies of the cat brain stem. volume 84, page 1136–1148, 2000.

[247] I. V. Tetko and A. E. P. Villa. A pattern grouping algorithm for analysis of spatiotemporal patterns in neuronal spike trains. volume 105, pages 15–24, 2001.

[248] P. Reinagel and R. Reid. Temporal coding of visual information in the thalamus. In *J. Neuroscience*, volume 20, pages 5392–5400. 2000.

[249] G.T. Buracas, A.M. Zador, M.R. DeWeese, and T.D. Albright. Efficient discrimination of temporal patterns by motion-sensitive neurons in primate visual cortex. In *Neuron*, volume 20, pages 959–69. 1998.

[250] J.A. Mazer, W.E. Vinje, J. McDermott, P.H. Schiller, and J.L. Gallant. Spatial frequency and orientation tuning dynamics in area v1. volume 99, pages 1645–50, 2002.

[251] T.J. Blanche, K. Koepsell, N. Swindale, and B. A. Olshausen. Predicting response variability in the primary visual cortex. 2008.

[252] H. A. Swadlow. Physiological properties of individual cerebral axons studied in vivo for as long as one year. volume 54, pages 1346–1362, 1985.

[253] H. A. Swadlow. Efferent neurons and suspected interneurons in motor cortex of the awake rabbit: Axonal properties, sensory receptive fields, and subthreshold synaptic inputs. volume 71, pages 437–453, 1994.

[254] H. A. Swadlow. Efferent neurons and suspected interneurons in binocular visual cortex of the awake rabbit: Receptive fields and binocular properties. volume 88, pages 1162–1187, 1988.

[255] H. A. Swadlow. Monitoring the excitability of neocortical efferent neurons to direct activation by extracellular current pulses. volume 68, pages 605–619, 1992.

[256] A.M. Thomson, J. Deuchars, and D.C. West. Single axon excitatory postsynaptic potentials in neocortical interneurons exhibit pronounced paired pulse facilitation. In *Neuroscience*, volume 54, pages 347–360. 1993.

[257] A. Reyes, R. Lujan, A. Rozov, N. Burnashev, P. Somogyi, and B. Sakmann. Target-cell-specific facilitation and depression in neocortical circuits. In *Nat Neurosci*, volume 1, pages 279–85. 1998.

[258] Henry Markram, Yun Wang, and Misha Tsodyks. Differential signaling via the same axon of neocortical pyramidal neurons. *Proceedings of the National Academy of Sciences of the United States of America*, 95(9):5323–5328, April 1998.

[259] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[260] Edward N Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.

[261] S. Sheik, E. Chicca, and G. Indiveri. Exploiting device mismatch in neuromorphic VLSI systems to implement axonal delays. In *International Joint Conference on Neural Networks, IJCNN 2012*, pages 1940–1945. IEEE, 2012.

[262] S. Sheik, F. Stefanini, E. Neftci, E. Chicca, and G. Indiveri. Systematic configuration and automatic tuning of neuromorphic systems. In *International Symposium on Circuits and Systems, (ISCAS), 2011*, pages 873–876. IEEE, May 2011.