Ariadna Study

Investigation of low energy Spiking Neural Networks based on temporal coding for scene classification

Final Report

Authors: Paolo Lunghi[1], Stefano Silvestrini[1], Dominik Dold[2], Gabriele Meoni[2], Dario Izzo[2]
Affiliation: [1]Politecnico di Milano, Aerospace Science and Technology Dept, [2]ESA ACT

Date: 24/08/2023

Contacts:

Paolo Lunghi
Tel: +39 02 2399 8041
Fax: +39 02 2399 8334
e-mail: paolo.lunghi@polimi.it

Leopold Summerer (Technical Officer)
Tel: +31(0)715654192
Fax: +31(0)715658018
e-mail: act@esa.int

Available on the ACT website
http://www.esa.int/act

Ariadna ID: 21/8601
Ariadna study type: Standard
Contract Number: 4000135881/21/NL/GLC/my

Ariadna Study

Investigation of low energy Spiking Neural Networks based on temporal coding for scene classification

Paolo Lunghi[1], Stefano Silvestrini[1], Dominik Dold[2], Gabriele Meoni[2], and Dario Izzo[2]

[1]Politecnico di Milano
[2]ESA ACT

24/08/2023

Contents

List of Acronyms

AC	Accumulate operation
ANN	Artificial Neural Network
API	Application Program Interface
BN	Batch Normalization
BNTT	Batch Normalization Through Time
BPTT	Backpropagation Through Time
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
EMAC	Equivalent Multiply-Accumulate operations
FLOP	Floating Point Operations
FLOPS	Floating Point Operations per Second
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HW	Hardware
IF	non-leaky Integrate and Fire (neuron model)
IFL	non-leaky Integrate and Fire Linear (neuron model)
LI	Leaky Integrator
LIF	Leaky Integrate and Fire (neuron model)
MAC	Multiply-Accumulate operation
MLP	Multi-Layer Perceptron
MVM	Matrix-Vector Multiplication
ROC	Rank Order Coding
R-STDP	Rewarded Spike Timing-Dependent Plasticity
SG	Surrogate Gradient
SNN	Spiking Neural Network
STDP	Spike Timing-Dependent Plasticity
SW	Software
TPU	Tensor Processing Unit
TTFS	Time To First Spike

1. Introduction

In recent years, interest in AI applications (notably ANNs) onboard satellites, has grown a lot. Potential applications in space systems include early detection of potential failures or catastrophic events [1, 2], feature detection and tracking in Guidance, Navigation, and Control systems [3, 4, 5], preprocessing of collected data by Eart Observation satellites to mitigate bandwidth requirements due to unmeaningful or corrupted data, advanced navigation and signal processing tasks [6, 7], and others. On the other hand, space systems are traditionally limited in computational and memory resources, a condition even more stressed due to the recent trend toward extreme miniaturization, i.e. CubeSats and other types of micro and pico satellites [2]. The adoption of highly energy efficient algorithm and computing could expand dramatically the autonomy of space missions.

Spiking Neural Networks (SNN) are highly attractive, due to their low-power and energy-efficient properties [8, 9, 10, 11, 12, 13, 14, 15, 16]. Such features are due to the brain-inspired architecture of this network, in which neurons communicate by means of discrete spikes. Neurons can be modeled in many different ways [17] but all of the models share a general behavior in which input event are somewhat accumulated along time, increasing an internal state called membrane voltage (mimicking the membrane voltage of biological neurons). Whenever the membrane voltage exceeds a certain threshold, a spike is released and the voltage is reset [16, 18]. Differently with respect to traditional ANNs, computation is performed only at the time a spike is received, making the overall activity inside the network inherently sparse [8, 10].

SNNs can be implemented on event-based neuromorphic hardware, designed to fully exploit such asynchronous, sparse computation paradigm to achieve solutions capable to outperform those based on ANNs in term of power consumption and energy efficiency [8, 9]. Nevertheless, the final performance is largely dependent on the combination of many parameters and design choices, such as the model of the neurons, the scheme of information encoding, target number of time steps at inference, and hardware implementation [8, 9, 10, 14, 19].

Nevertheless, state-of-the-art SNNs struggle to scale to very deep models, due to lacking of suitable training algorithms. Looking for biologically plausible mechanisms, researchers proposed unsupervised training methods like Spike Timing-Dependent Platicity (STDP), a form of Hebbian learning where weight changes depend on the relative timing between pre and postsynaptic action potentials [20, 21, 22]. Using STDP, the network can successfully extract frequently occurring visual features. However, an unsupervised learning rule alone is not sufficient for decision making. To cope with this issue, Rewarded STDP (R-STDP) has been proposed [23, 24], a reinforcement learning rule designed to modulate STDP to match a desired output. Nevertheless, STDP-based networks still have to demonstrate their effectiveness in complex tasks.

Some approaches have been studied that allow conversion between deep ANNs and SNNs [12, 14, 25]. In many of these cases, the conversion process is based on the close correlation between the activation rate of spiking neurons and the activation of Rectified Linear Units (ReLUs) in standard ANNs. Thanks to these methods, rate-based conversion between ANNs and SNNs can be done with minimal loss of accuracy [8, 14, 25]. However, the use of rate-based SNNs usually requires high fire-rates and a high number of timesteps to provide acceptable accuracy [12, 18]. In this respect, this approach seems to bring real benefits in terms of energy efficiency only for event-based datasets [9]. On the contrary, for standard static images, which represent the data of interest for many remote sensing applications, the gain in terms of energy savings seems to be reduced for complex datasets due to the higher number of timesteps required, which also leads to higher processing latencies [8, 9].

Methods based on temporal coding, also called *latency-based* methods, might offer more promising trade-offs [10, 16, 12, 13, 15, 19]. Such approaches encode information in the fire time of neurons. According to the Time-To-First-Spike (TTFS) coding, the more a neuron is activated, the shorter is its firing delay, and more significant the associated transmitted information [10, 16, 12, 13]. For classification tasks, Rank Order Coding (ROC) is even more simple, for the attribution class corresponds to the ouput neuron that fires first, regardless of the exact spike time. In such encoding schemes, neurons can be even forced to fire once at most during an inference, making SNN models based on

temporal coding extremely attractive for energy-constrained applications. In a benchmark comparison among different coding schemes, latency-based methods showed the highest potential performances in terms of accuracy, latency, power consumption and number of synaptic operation, albeit with some potential lack of robustness in case of input or synaptic noise [26]. Some energy efficiency benefits of time-coding based SNNs for static data are shown in [16] for the MNIST dataset on the BrainScaleS-2 neuromorphic processor. However, the applications of SNNs for space applications are still limited [27, 28, 29, 30], and the ability of these models to cope with complex features such as those included in scene classification datasets still awaits a convincing demonstration.

Recently, novel training method like Surrogate Gradient (SG) [31, 18, 32, 33, 34] and Backpropagation Through Time (BPTT) [35, 36, 37, 15, 13, 10, 19] showed promising result in successfully training deeper networks, but still an effective demonstration on latency-based SNNs on more complex tasks is still missing.

1.1. Study objectives

This study aims to perform a preliminary investigation of the potential benefits of Spiking Neural Networks (SNNs) based on temporal coding for onboard Artificial Intelligence (AI) applications, considering the case study of scene classification. To achieve this goal, state of the art SNN models are to be compared in terms of accuracy and complexity (here considered as the number of synaptic operations, number of spikes per neuron, and others) on a scene classification task, using the EuroSAT RGB dataset (see the next Section). To this aim, proper training algorithms for the SNN models shall be also evaluated and selected. Eventually, the aim is to establish a method to perform hardware-agnostic, relative comparison of the computational load required by different architectures, both SNNs and ANNs. The results of the analysis will highlight the possible advantages and drawbacks of SNN models compared to ANNs, which represent the state of the art for scene classification.

1.2. Case study: EuroSAT dataset

EuroSAT is a reference dataset for scene classification representative of a plausible use case in the Earth Observation field, i.e. land use classification [38]. It includes multispectral imagery provided by the Sentinel-2A satellite, divided into 13 bands over 10 different land use and land cover classes. An RGB version of the dataset is also provided.

The activity presented in this report focuses on the RGB EuroSAT dataset. Each image is then an 8 bit, $3 \times 64 \times 64$ px (c, h, w) in size tensor. The dataset consists in 27 000 images, divided in 10 classes each one represented by a number of samples between 2000 and 3000. Some samples for each class in the dataset are shown in Fig. 1. A 70/20/10 (training, validation, test) subdivision has been adopted for the training and cross-validation of all the models tested in this activity. Random horizontal and vertical flip have been adopted as only data augmentation at training time.

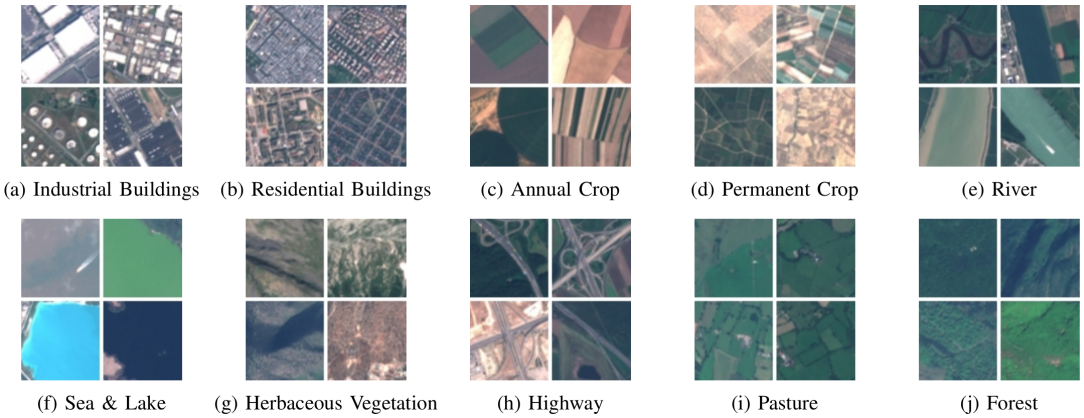


Figure 1: RGB EuroSAT dataset samples from [38].

2. Enabling latency-based SNNs with backpropagation

In this section the training methods considered in this work are presented. The following criteria have been followed in the seclection:

- suitability for latency-based networks, which constitute the main subject of the study;
- flexibility to handle different neuron models and encoding schemes;
- possibility to fast-prototyping benchmark cases;
- compatibility with existing network training libraries (i.e. PyTorch or Tensoflow) to leverage efficient computation and optimization tools.

Two alternatives, emerged in recent years as most prominent methods to train latency-based networks, have been considered: minimization of spiking times error and Backpropagation Through Time (BPTT) with Surrogate Gradient (SG) training.

2.1. Backpropagation Through Time (BPTT)

Once the graph of a discrete time SNN is unrolled along the time dimension, SNNs works essentially as Recurrent Neural Networks (RNN). The most common training methods for such networks is the Backpropagation Through Time (BPTT), consisting in the direct application of error-backpropagation to this unrolled graph. Nevertheless, and differently w.r.t. RNNs, the intrinsically discontinuous nature of the spikes prevents a correct flowing of the gradients through the network in the backward pass, making the application of standard BPTT to SNNs infeasible. To overcome this limitation, two popular approaches are considered: optimization of spike times and Surrogate Gradient (SG) techniques.

2.2. Optimization of spike times

This training approach considers spike times instead of spikes themselves as information-carrying quantity. This is achieved by relating the time of any spike, in a differentiable way, to the times of the presynaptic spikes that had a causal influence on its generation. In this way, the (discontinuous) derivation of the thresholding activation function is no longer required. Instead, dealing with a continuous representation make gradient descent by means of backpropagation feasible.

This training approach allows a high degree of control on the internal dynamics of a network, down to the single spike time, which would be impossible in other training methods. On the other hand, it is heavy tailored on the specific network: whenever neuron model, loss function, regularization loss functions (if any), and other parameters are changed, the formulation of the derivatives could need to be rewritten from scratch. As example, in [36], is formulated for non-leaky integrate-and-fire with exponentially decaying synaptic current neuron (LIF, see Sec. 3.3; [13] applies the technique to non-leaky integrate-and-fire neurons with instantaneous synapse (IAF, see Sec. 3.1); [37] considers non-leaky integrate-and-fire neurons with step-wise constant synaptic current (IFL, see Sec. 3.2); [19] formulates the problem for alpha-synaptic function neurons. Moreover, optimization of spike times is intrinsically constrained to stick to latency encoding, usually with neurons forced to spike once at most.

Remark. *Due to this lack of flexibility, optimization of spike times was not selected as the main methodology to carry out testing activity in this work. Nevertheless, one test case using this technique has been included for comparison purposes.*

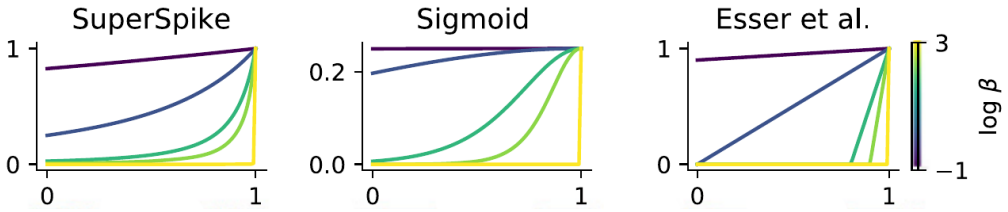


Figure 2: Examples of surrogate gradient function from [32]: *SuperSpike* [31], which is the derivative of a fast sigmoid function: $h(x) = (\beta|x| + 1)^{-2}$; the derivative of a standard *Sigmoid* functiton: $h(x) = s(x)(1 - s(x))$ with the sigmoid function $s(x) = (1 + \exp(-\beta x))^{-1}$; *Esser et al.*, a piece-wise linear function [39, 40]: $h(x) = \max(0, 1.0 - \beta|x|)$. The trend for each gradient function is shown for different values of the hyperparameter β.

2.3. Surrogate gradient (SG)

Surrogate Gradient (SG) training has emerged as a popular solution for enablig training SNNs end-to-end with BPTT. Neuronal spiking dynamics lacks of a continuous gradient, preventing them to be directly optimized by traditional backpropagation algorithms. To circumvent this limitations, in the SG method the actual derivative of a spike, which appears in the analytic expressions of the gradients, is replaced by any well-behaved function. There are many possible choices of such surrogate derivatives, making the resulting surrogate gradient not unique (differently to the unique true gradient of a function). Examples of possible choices in shown in Fig. 2. Several studies have successfully applied different types of surrogate derivatives to various problem sets [39, 40, 41, 34, 42, 31], suggesting that the methods does not crucially depends on the specific choice of the surrogate derivative. Such hypothesis was proven in [32], where it is shown that SG is robust to the shape of the surrogate derivative, but also to changes in the loss function, input paradigms, and dataset. Such robustness make SG methods very attractive to application to SNNs:

- the use of SG in standard frameworks for backpropagation training (e.g. PyTorch, TensorFlow...) requires only the surrogate gradient function to be implemented in a custom backward function. Then, the reminder of the network and the training process (loss, regularization loss, etc.) can be formulated with standard ANN building blocks, maintaining the coding effort to a minimum;
- different neuron models can be easily compared with no additional effort, being the formulation independent on the specific neuron model;
- the training method is not constrained to a specific coding scheme, and can be applied to both rate and latency based encoding.

Remark. *Due to the possibility of fast formulation and easy comparison between different architectures and neuron models, Surrogate Gradient training has been selected as main training method in this work. If not explicitly stated, test cases are trained with Surrogate Gradient, in its SuperSpike [31] variant.*

For rate-based networks, the method proved to be sensitive to weights initialization [33], but such limitation is still to be assessed for latency-based systems. Moreover, even if most of the works cited in this section applied SG to rate-based networks, the method is not constrained to a specific coding, once the possibility to the gradient to flow is granted. To this purpose, a special TTFS readout layer (see Sec. 3.5.2) has been developed to convert output spikes to spike times in a differentiable way, enabling training with SG with latency-based encoding networks.

3. Neuron Models

The most peculiar feature of SNNs is that the neurons possess temporal dynamics: typically, an electrical analogy is used to describe their behavior. Each neuron has a voltage potential that builds up depending on the input current that it receives. The input current is generally triggered by the spikes the neuron receives. There are numerous neural architectures that combine these notions into a set of mathematical equations [17]. Most of them are summarized in Fig. 3.

Models	biophysically meaningful	tonic spiking	phasic spiking	tonic bursting	phasic bursting	mixed mode	spike frequency adaptation	class 1 excitable	class 2 excitable	spike latency	subthreshold oscillations	resonator	integrator	rebound spike	rebound burst	threshold variability	bistability	DAP	accommodation	inhibition-indiced spiking	inhibition-induced bursting	chaos	# of FLOPS
integrate-and-fire	-	+	-	-	-	-	-	+	-	-	-	-	+	-	-	-	-	-	-	-	-	-	5
integrate-and-fire with adapt.	-	+	-	-	-	-	+	+	-	-	-	-	+	-	-	-	-	+	-	-	-	-	10
integrate-and-fire-or-burst	-	+	+		+	-	+	+	-	-	-	-	+	+	+	-	+	+	-	-	-		13
resonate-and-fire	-	+	+	-	-	-	-	+	+	-	+	+	+	+	-	-	+	+	+	-	-	+	10
quadratic integrate-and-fire	-	+	-	-	-	-	-	+	-	+	-	-	+	-	-	+	+	-	-	-	-	-	7
Izhikevich (2003)	-	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	13
FitzHugh-Nagumo	-	+	+	-		-	-	+	-	+	+	+	-	+	-	+	+	-	+	+	-	-	72
Hindmarsh-Rose	-	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	120
Morris-Lecar	+	+	+	-		-	-	+	+	+	+	+	+	+		+	+	-	+	+	-	-	600
Wilson	-	+	+	+			+	+	+	+	+	+	+	+	+	+		+	+				180
Hodgkin-Huxley	+	+	+	+			+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	1200

Figure 3: Comparison of the neuro-computational properties of spiking and bursting models; "# of FLOPS" is an approximate number of floating point operations (addition, multiplication, etc.) needed to simulate the model during a 1ms time span. Each empty square indicates the property that the model should exhibit in principle (in theory) if the parameters are chosen appropriately, but the author of [17] failed to find the parameters within a reasonable period of time. Figure taken from [17].

Remark. *In this activity, biological plausibility was not included in the criteria considered in the formulation of the test cases. Hence, only the most simple neuron models, among the wide list of possibilities, were considered, for energy efficiency is the main parameter of interest in the adoption of SNNs in space applications.*

In the following, the models considered in this study are briefly presented.

3.1. Integrate and Fire (IAF)

The IF neuron model assumes that spike initiation is governed by a voltage threshold. When the synaptic membrane reaches and exceeds a certain threshold, the neuron fires a spike and the membrane is set back to the resting voltage V_{rest}. In mathematical terms, its simplest form reads:

$$C\frac{dV(t)}{dt} = i(t) \tag{3.1}$$

The first implementation used in this work entails a direct synapse, where input spikes are just integrated directly. In the discrete implementation, the spikes are just weighted Kronecker's deltas, thus the are injected into the neuron and summed up to the membrane voltage. Therefore, the voltage trend is step-wise constant, as shown in Figure 4.

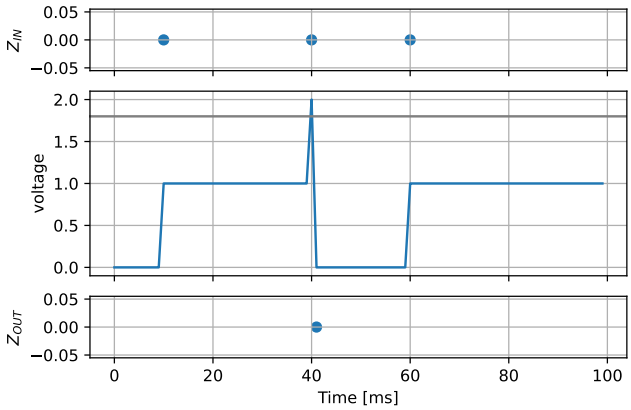
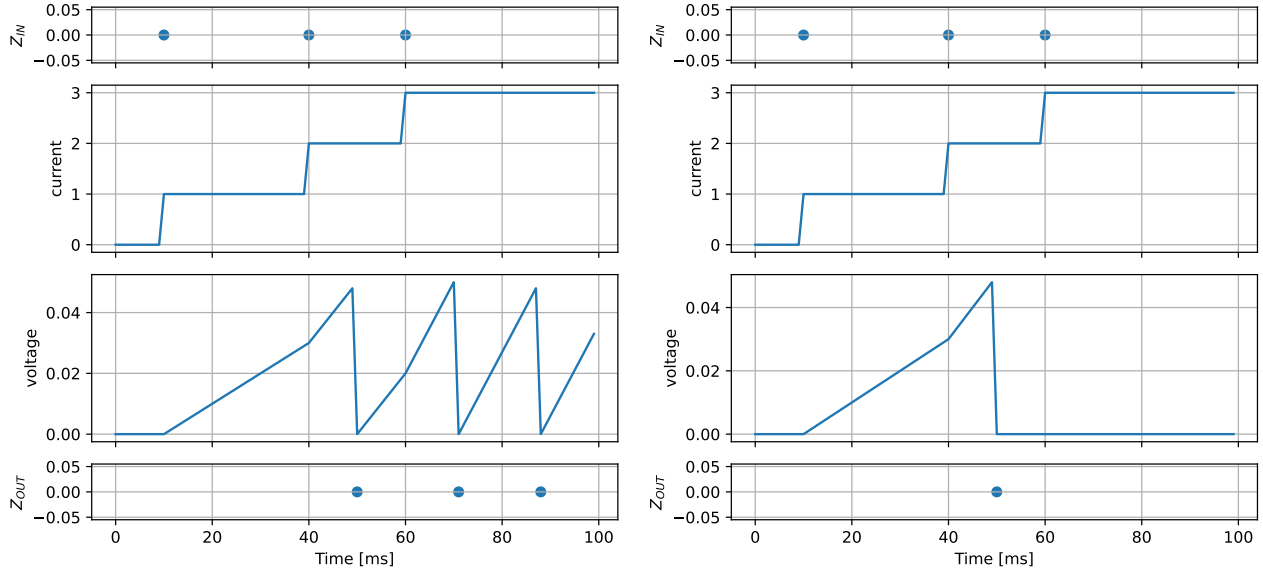


Figure 4: Example of membrane voltage, input and output spikes in a Non-leaky Integrate and Fire neuron with direct synapse. In this neuron model, there is no current state.

3.2. IFL

The non-leaky itegrate and fire linear neuron (IFL) is a slight modification of the direct synapse integrate and fire neuron (IF). In this model the synapse receives the (weighted) input spikes and updates the post-synaptic current i with the formula: $\frac{di(t)}{dt} = \sum_t S(t)$. In the discrete version, since the input spikes are just weighted Kronecker's deltas, they are just summed up to the post-synaptic current yielding a linear behavior of the membrane potential voltage, as shown in Figure 5.



(a) IFL neuron.

(b) IFLOnce neuron.

Figure 5: Non-leay Integrate and Fire with stepwise constant synapse. Example of input spikes, current, voltage, and output spikes. 5a) standard version of IFL neuron; 5b) IFL which fires once at most.

3.3. LIF

The LIF neuron is a slightly modified version of the IF neuron model. Indeed, it entails an exponential decrease in membrane potential when not excited. The membrane charges and discharges exponentially in response to injected current. The differential equation governing such behavior can be written as:

$$C\frac{dV(t)}{dt} + \lambda V(t) = i(t) \tag{3.2}$$

where λ is the leak conductance and V is again the membrane potential with respect to the rest value. Figure 6 shows how exponential relaxation of $V(t)$ to a steady state value follows current injection, where 0 is the initial membrane potential and reset value.

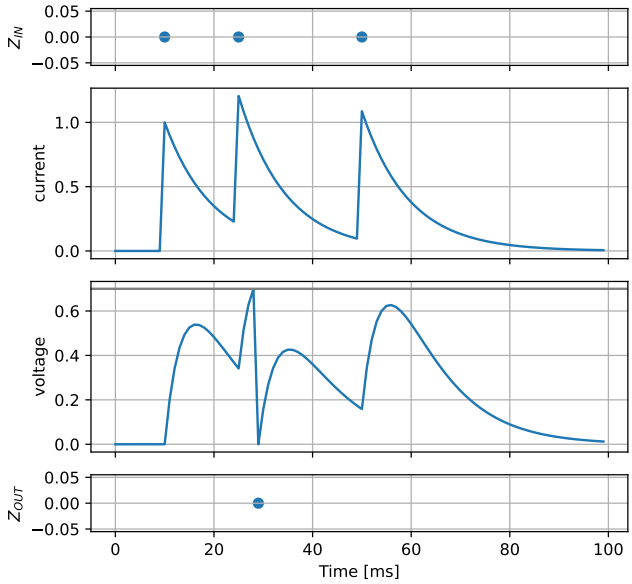


Figure 6: LIF neuron. Example of input spikes, current, voltage, and output spikes.

3.4. Special features

All the spiking layers can be equipped with additional special features, detailed in this Section.

3.4.1. Neurons Spiking Once at most

All the types of neuron can be forced to spike once at most (by setting a large refractory time after the spike or just by design). Such feature is attractive in the pursuit of an energy-efficient network implementation, mainly for two reasons:

1. the number of spikes emitted at inference is largely limited;
2. once a neuron has emitted a spike, the related memory can be freed, limiting even more the netowrk memory footprint.

3.4.2. Recurrent spiking layers

Working along time, spiking layers can be *recurrent*: their output can be mapped as input to the layer itself in the subsequent time step. Such recursion can we weighted, storing additional information in the process as in standard Recurrent Neural Networks.

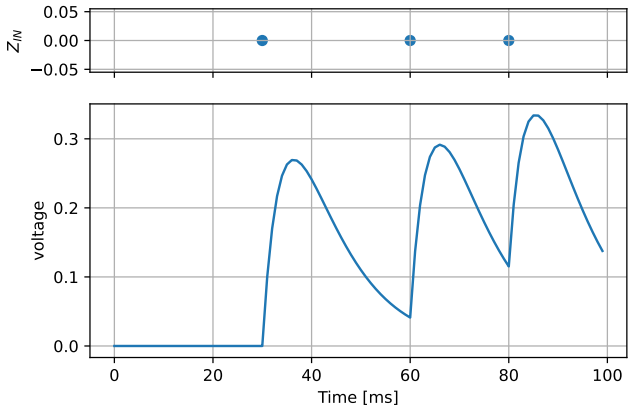


Figure 7: LI neuron. Top: input spikes. Bottom: LI neuron voltage. The layer act as a LIF neuron except that no spike is emitted.

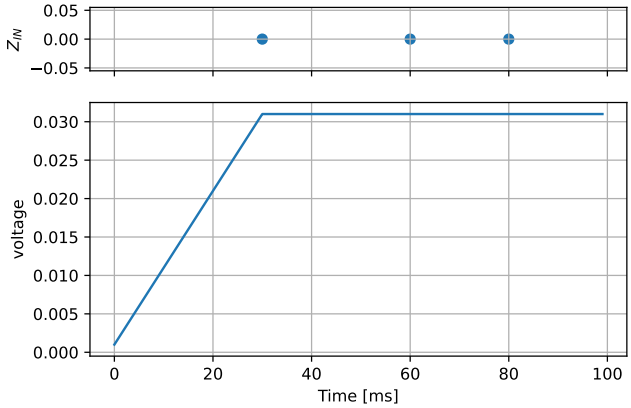


Figure 8: TTFS readout layer. Top: input spikes. Bottom: neuron internal state. As the first input spike is received, the time integration stops, fixing the value to the time-to-first-spike. Subsequent spikes do not further alter the value of internal state.

3.5. Output layers

3.5.1. Leaky Integrator (LI)

A Leaky Integrator works in the same way as a LIF neuron, except for the fact that it is a non-spiking layer: there is no threshold and there is no emission of spikes as output. Such type of layer is usually placed as last layer at the network output for rate-base decoding schemes (see Sec. 5). Its behavior is visually represented in Fig. 7.

3.5.2. TTFS readout

The function of a TTFS readout layer is to output the time of the first received spike in a way that preserves automatic differentiation, to enable backpropagation with latency decoding (see Sec.5). The layers acts as a simple time integrator: the integration stops whenever the first input spike is received. Then, the value of this internal state (hereby referred to voltage) at the last time step coincides with the time the spike has been received. Figure 8 shows the typical behavior of such layer. Obviously, to perform such readout operation the layer must be placed just after a spiking layer, and 1-to-1 synaptic connection for each neuron of the previous layer is required.

4. Input encoding

The transition between dense data and sparse spiking patterns requires a coding mechanism for input coding and output decoding. One of the distinguishing features of spiking neural networks is that they operate on temporal data encoded as spikes. Common datasets in machine learning do not use such an encoding and therefore make an encoding step necessary. For what concerns the input coding, the data can be transformed from dense to sparse spikes in different ways, among which the most used are:

- Rate coding: it converts the input intensity into a firing rate or spike count;
- Temporal (or latency) coding: it converts the input intensity to a spike time or relative spike time.

Similarly, in output decoding, the data can be transformed from sparse spikes to network output (such as classification class) in different ways, among which the most used are:

- Rate coding: it selects the output neuron with the highest firing rate, or spike count, as the predicted class;
- Temporal (or latency) coding: it selects the output neuron that fires first, or before a given threshold time, as the predicted class

Roughly speaking, the current literature agrees on specific advantages for both the coding techniques. On one hand, the rate coding is more error tolerant given the reduced sparsity of the neuron activation. Moreover, the accuracy and learning convergence have shown superior results in rate-based applications so far. On the other hand, given the inherent sparsity of the encoding-decoding scheme, latency-based approaches tend to outperform the rate-based architectures in inference, training speed and, above all, power consumption [26].

4.1. Rate based encoding

4.1.1. Constant Current LIF

Encodes input currents as fixed (constant) voltage currents, and simulates the spikes that occur during a number of timesteps. Here we choose to treat the grayscale value of an MNIST image as a constant current to produce input spikes to the rest of the network. As can be seen from the spike raster plot, this kind of encoding does not produce spike patterns which are necessarily biologically realistic. We could rectify this situation by employing cells with varying thresholds and a finer integration time step.

4.1.2. Poisson

This rate-based encoder is formulated under the *independent spike hypothesis*, which is the assumption that if the precise spiking time of a neuron is not important, and all the information is carried by the spike frequency, then each spike of each neuron can be generated randomly, independently from the others. Spikes are then produced by a *Poisson random generator*, set to hit a target spike rate proportional to the input to be encoded [43, 44]. Figure 11 shows a raster plot of an entry from the EuroSAT dataset, processed by a Poisson encoder.

4.2. Latency encoding

Latency encoders codify the information in the spike times. Usually, the most siginificant and/or higher value information correspond to an earlier time of the spike.

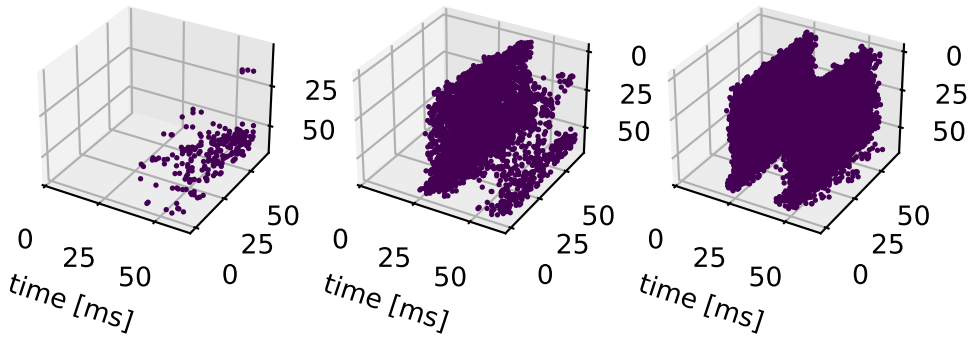


Figure 9: Example of encoding of an RGB Eurosat input image. Each channel is shown as separate image.

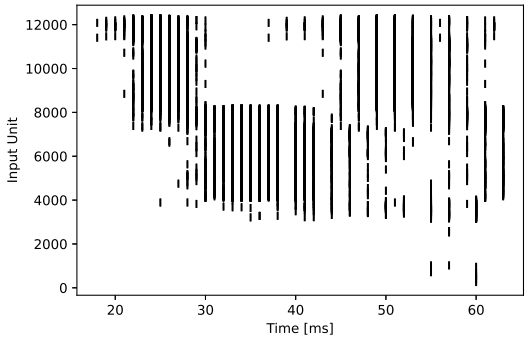


Figure 10: Raster plot for a constant current LIF encoder.

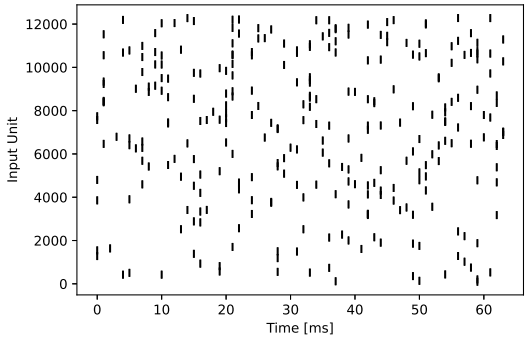


Figure 11: Raster plot for a Poisson encoder.

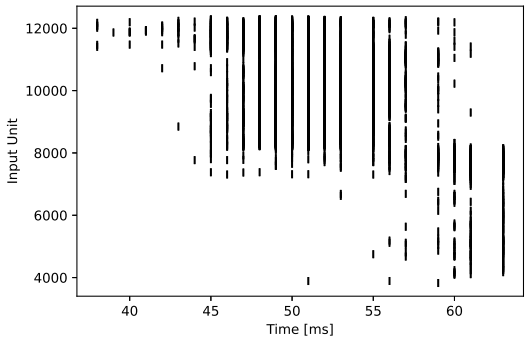


Figure 12: Raster plot for a latency LIF encoder.

4.2.1. Linear TTFS encoder

The input value x_i is linearly mapped to a spike time $0 \leq t_i \leq t_{\max}$ with the following equation:

$$t_i = t_{\max}\left(1 - \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}\right) \tag{4.1}$$

where the input is assumed to be bounded between $x_{\min}$ and $x_{\max}$, and $t_{\max}$ is a hyperparameter. The higher the value, the earlier the neuron spikes.

4.2.2. Constant Current IAF (one spike at most)

In this type of encoder, the input value is fed as a constant current into a layer of Integrate-and-Fire neurons (see Sec. 3.1 at page 9), set to spike once at most. In this way, the higher the input, the sooner the correspondent neuron spikes, in a TTFS scheme. This scheme works well as long as the input is positive (i.e. images): in fact, negative inputs would lead to silent neurons.

4.2.3. Latency LIF encoder

Similarly to Constant current IFL (Sec. 4.2.2) the input is directly fed to a layer of LIF neurons, with an infinite refractory time. In this way, the neurons spike only once at most, and the higher input value lead to earlier spiking times.

4.3. Convolutional learnable encoder

In an ideal implementation, the encoders expounded above generate spikes at continuous times, capable to encode information with infinite precision. But in a digital implementation, spikes cannot happen at arbitrary times, but only at discrete time steps. This can potentially lead to loss of information, especially if low latency (and consequently a reduced number of time steps) is pursued. Let's take the case of a linear TTFS encoder (see Sec. 4.2.1): a typical RGB input entails 3 channels in which the information is represented with 8 bit, thus 256 levels. At encoding, if for example $\Delta t = 1\,\text{ms}$ and $t_{\max} = 32\,\text{ms}$ are selected, if each pixel in each channel is mapped to a single neuron, only 32 levels are available to encode for the pixel intensity. This potential problems applies also to other types of encoders, both rate and latency based. In fact, even if it could be a viable way to compress information, there no guarantee that the process would be lossless. To cope with this issue, the first, most obvious option would be an appropriate selection of the hyperparameters to ensure the necessary number of time steps to completely represent the input. Nevertheless, such solution would potentially lead to large number of steps, with consequently long latency and increased energy consumption. The other possibility is to first apply a convolution to the original input, with appropriate large number of output channels, and then feed the convoluted input to the desired encoder. In this way, the

original information is spread over a higher number of channels, allowing a complete representation of the input even in a reduced number of time steps. The weight of the convolution filters could also be optimized by including them in the training. This solution can be combined with every type of encoder, to achieve a learnable optimal encoder.

5. Output decoding

In this section, the decoders adopted in this work are expounded. The encoder/decoder coupling is not predetermined: a latency-based encoding can be mixed with a rate-based decoder and vice versa. The network type (latency/rate) is determined by the decoder.

Remark. *In this work, being the analyzed benchmark problem essentially a classification task, latency-based decoding follows a* Rank Order Coding (ROC) *scheme.*

In fact, while the time to first spike is used as classification score for the class represented by a certain output neuron, the precise timing of the spike is not considered, and only relative differences count, being the scores always normalized by Softmax or LogSoftmax functions.

5.1. Last timestamp logarithmic voltage

Rate-based decoding. To apply this decoding, the last layer of the architecture should consists of N output Leaky-Integrate (LI) neurons, where N is the number of classes that should be distinguished. Indeed, each neuron is assigned to a specific class. Decoding is, then, performed by applying a logarithmic softmax to the voltage values of the various neurons of the architecture in the last timestep and picking the maximum, as shown in Eq. (5.1):

$$c = \arg\max_i \Big(\text{LogSoftmax}\big(V_i(t = T)\big) \Big) \tag{5.1}$$

5.2. Maximum voltage

Rate-based decoding. For the i-th class, the score is represented by the maximum value assumed by the potential of the related i-th output neuron V_i along the inference time, normalized then by a softmax function. The decoder equation is represented in Eq. (5.2). It is designed to be coupled with a LI output layer (Sec. 3.5.1).

$$c = \arg\max_i \Big(\text{Softmax}\big(\max V_i(t)\big) \Big) \tag{5.2}$$

5.3. Maximum logarithmic voltage

Rate-based decoding. For the i-th class, the score is represented by the maximum value assumed by the potential of the related i-th output neuron V_i along the inference time. The score is then processed by a logarithmic softmax. The decoder equation is represented in Eq. (5.3). It is designed to be coupled with a LI output layer (Sec. 3.5.1), and it works with a negative log likelihood loss function[1].

$$c = \arg\max_i \Big(\text{LogSoftmax}\big(\max V_i(t)\big) \Big) \tag{5.3}$$

5.4. Negative Time-To-First-Spike

Latency decoding. For the i-th class, the score is represented by the negative of the time-to-first-spike of the related i-th output neuron τ_i. The score is then processed by a logarithmic softmax. The decoder equation is represented in Eq. (5.4). It works with a negative log likelihood loss function[2]. In this work, it is used for models trained with BPTT as in [36].

$$c = \arg\max_i \big(\text{LogSoftmax}(-\tau_i) \big) \tag{5.4}$$

[1] `https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html` – Last visited: 27/06/2023.
[2] Ibid.

5.5. Logarithmic inverse Time-To-First-Spike

Latency decoding. For the i-th class, the score is represented by the inverse of the time-to-first-spike of the related i-th output neuron τ_i. The decoder equation is represented in Eq. (5.5). It works with a negative log likelihood loss function[3]and takes as input the times at which the first spikes are emitted by the output layer, obtained by means of a TTFS readout layer (Sec. 3.5.2) or other methods. In this works, it is the default decoder for latency-based test cases.

$$c = \arg\max_i \big(\text{LogSoftmax}(1/\tau_i)\big) \tag{5.5}$$

[3]Ibid.

6. Regularization

Three network normalization schemes have been tested on latency networks:

- Target output time (Sakemi 2021, [37]);
- Sum of weights (Stanojevic 2021, [45]);
- Batch Normalization Through Time (Kim 2021, [46].

6.1. Target output time

The regularization, originally proposed in [37] for improve training in SNNs based on rank order coding with Backpropagation Through Time, consists in the modification of the cost function is the form:

$$\mathcal{C}(\boldsymbol{t}^{(M)}, \boldsymbol{k}, \boldsymbol{w}) = \mathcal{L}(\boldsymbol{t}^{(M)}, \boldsymbol{k}, \boldsymbol{w}) + \frac{\gamma}{2}\mathcal{R}(\boldsymbol{t}^{(M)}) \tag{6.1}$$

where $\mathcal{C}$ is the cost function, $\mathcal{L}$ is the training loss function, $\boldsymbol{k}$ and $\boldsymbol{w}$ are respectively the training labels and the network weights. The temporal penalty term $\mathcal{R}(\boldsymbol{t}^{(M)})$, weighted by the term γ, is defined by the total difference between the spike timing of the output neurons $\boldsymbol{t}^{(M)}$ and the timing of a reference spike $t^{(\text{ref})}$:

$$\mathcal{R}(\boldsymbol{t}^{(M)}) = \sum_{i=1}^{N} \left(t_i^{(M)} - t^{(\text{ref})}\right)^2 \tag{6.2}$$

where N is the number of neurons in the output layer (which coincides with the number of the classes). During the training, this penalty term forces all the output neurons to spike improving the gradient flow in backpropagation. At inference, this adds no additional activity in the network, since the computation ends as the first output neuron spikes (before $t^{(\text{ref})}$). The effectiveness of this regularization scheme is here assessed with SG training.

6.2. Regularization loss based on the sum of synaptic weights

In this type of regularization a penalization term, based on the sum of the weights in each neuron's receptive field, is added to the loss to counteract dead neurons. In the scheme proposed in [45] such penalization term takes the form:

$$\mathcal{L}_{\text{silent}} = k \sum_i \max(0,\ -\sum_j w_{ij}) \tag{6.3}$$

where k is an hyperparameter, and w_{ij} are the weights at each neuron input. Such regularizer would ensure that the summation of input weights to a neuron does not drop below zero so as to avoid silent neurons. In [45], it is used with a ANN/SNN conversion approach, still in a latency-based network together with a neuron model very similar to the IFL discussed in Sec. 3.2. In this work, its efficacy is assessed with SG training.

A very similar penalty term is adopted in [47], associated with BPTT training:

$$\mathcal{L}_{\text{silent}} = k \sum_i \max(0,\ 1 - \sum_j w_{ij}) \tag{6.4}$$

in this form, the regularizer assures that the sum of the weights of the i-th neuron is above 1, which ensures that a neuron spikes if all its input neurons spike. Instead of 1, a tunable threshold can be adopted like a hyperparameter. Such regularized has been adopted for test cases based on BPTT training (see Sec. 9.1).

6.3. Batch normalization through time (BNTT)

Batch Normalization [48] is one of the most adopted regularization techniques used to improve and accelerate training in ANNs. However, when applied to SNNs and Surrogate Gradient training in particular, marginal improvement have been observed [49]. To cope with this limitation, in [46] Batch Normalization Through Time (BNTT) has been proposed. BNTT decouples the hyperparameters of a BN layer (notably the output scaling parameter γ) along time, and make them learnable at training time. This allows the BNTT layer to learn input time pattern, enabling low-latency and low-energy performances.

Aside of this feature, BNTT acts as a standard BN layer. At training, given a mini-batch $\mathcal{B} = \{x_{1,\dots,m}\}$, the mean and variace of $\mathcal{B}$ are computed as:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{b=1}^{m} x_b; \qquad \sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{b=1}^{m}(x_b - \mu_{\mathcal{B}})^2. \tag{6.5}$$

Then the input features are normalized with:

$$\hat{x}_b = \frac{x_b - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{6.6}$$

where ϵ is a small constant to avoid numerical instability. In standard BN, at this point the normalized input is fed to the next layer after an affine transformation:

$$\mathrm{BN}(x_i) = \gamma\hat{x}_i + \beta \tag{6.7}$$

where γ and β are learnable parameters. Global statistics μ and σ^2 are learned along batches by means of exponential moving average. At inference time, the stored global statistics are used to normalize the input.

Conversely, in SNNs, the input has an additional dimension along time. In BNTT, a different value of γ^t is used for each time step t, and μ^t and $(\sigma^2)^t$ are computed separately for each t as well. Moreover, the BNTT layer is applied in convolutional layer just after the convolution and before the spiking layer. This allow the imposition of $\beta = 0$, since the offset would be just redundant with the learnable bias in the convolution. Then:

$$\mathrm{BNTT}(x_i^t) = \gamma^t\hat{x}_i^t \tag{6.8}$$

From the energy consumption perspective, the placement of the BNTT layer just before spiking neurons would allow, in a trained network at inference time, to incorporate the normalization operations in the convolution weights, without further computational load. BNTT has been originally developed in [46] for rate-based networks with LIF neurons, trained from scratch with SG. In this work, its effectiveness in latency based network with IFL neurons is tested. The normalization of the input can be applied across different dimensions of the minibatch. Two variants have been tested here: *neuron-wise* BNTT, in which statistics and hyperparameters are computed for each single neuron, and *spatial* BNTT, in which statistic are computed for each channel in the convolution (so there is a dedicated value of μ, β, γ for each layer, for each channel, for each time step).

7. SNN models

Spiking Neural Network models adopted in this work do not differ with respect to regular ANNs, except for the activation functions, replaced by layers of spiking neurons. Multilayer Perceptrons (MLP) and Convolutional Neural Networks (CNN) are the building blocks used in this activity to define the test cases. A MLP with Limited Receptive Field has also been developed to circumvent some possible limitations of neuromorphic hardware in implementing convolution (even if SNNs hardware tests are not directly included in this activity). In this section, only significant differences w.r.t. standard Neural Network models are detailed.

7.1. Multilayer Perceptrons

The most simple ANN architecture, Spiking MLPs up to 3 hidden layer in size have been tested.

7.2. Convolutional Spiking Neural Networks

State-of-the-art performances in image classification tasks are achieved by means of Convolutional Neural Networks. A benchmark architecture of a Spiking CNN has been formulated to be adopted in numerical tests. The use of a common baseline structure for test cases is useful to highlight the relative impact of single changes. This allows to easily compare SNN-specific variations: neuron model, encoder/decoder models, neuron hyperparamentes, and so on.

Such benchmark architecture is shown in Fig. 13. A VGG-style general architecture has been selected: it consists in a learnable convolutional encoder (see Sec. 4.3), followed by two convolutional layers. Except at the encoder, at each convolution spatial dimensions half in size, while the number of filters is doubled. Two fully connected layers (a hidden layer and the output layer) are placed on top of the network. The reduction in size along the spatial dimensions (height and width) is achieved by means of a stride value $s = 2$ in the convolution, a solution that proved to be more efficient at training SNNs in preliminary tests. Nevertheless, also models with classic max pooling has been tested. Variations with different number of layers, different neurons, types of regularization, size of layers, and encoding/decoding styles have been studied.

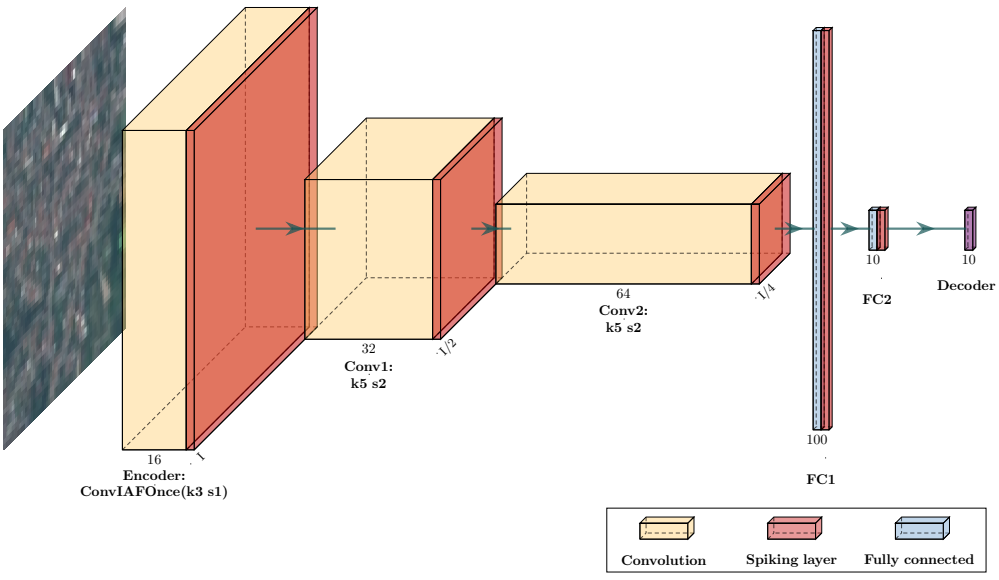


Figure 13: Benchmark convolutional architecture.

7.3. Multilayer perceptron with limited receptive fields

Convolutional Neural Network models are one of the most natural choice for the processing of images for artificial models, given the high compatibility with hardware such as Graphics Processing Units (GPUs) or other devices. This is not the case for most of neuromorphic hardware. Indeed, mixed-signals chips as BrainScale2 [50] are incompatible with convolution, given the design of the analog crossbar array hardware performing Multiply and ACcumulation (MAC) operations. Even for digital neuromorphic hardware devices, which could theoretically support convolution, this is not always the case. For instance, in the case of Loihi [51], the lack of support is due to the design of the software Application Program Interface (API). In many of these cases, to infer convolutional models on neuromorphic hardware, it is necessary to unroll the convolutional operation, leading to a memory inefficient implementation.

Because of that, we decided to include Multi-Layer-Perceptron (MLP) models among the models to test and benchmark, which represents a more hardware compatible choice compared to convolutional spiking models. Generally the models are made by two or more cascaded MLP layers, each one having LIF neurons as activations. The only exception is the last layer, which include LI neurons to enable the use of *Maximum last timestamp logarithmic voltage decoding*, detailed in Sec. 5.1.

To emulate the behavior of convolutional models, which process images with a limited receptive fields, we artificially limited the receptive field of the first MLP layer by multiplying its weights by a matrix that masks the connections outside of a specific receptive field. An example of MLP-LRF model is shown in Fig. 14. Furthermore, to potentially increase the performance, some of the models were tested by making the LIF and LI neurons of each layers trainable. To this aim, synaptic and membrane constants were randomly initialized at the beginning of training by using a normalized distribution. In case of trainable LIF neurons, threshold voltage were also made trainable and initialized by using a user-specified initial value for each layer.

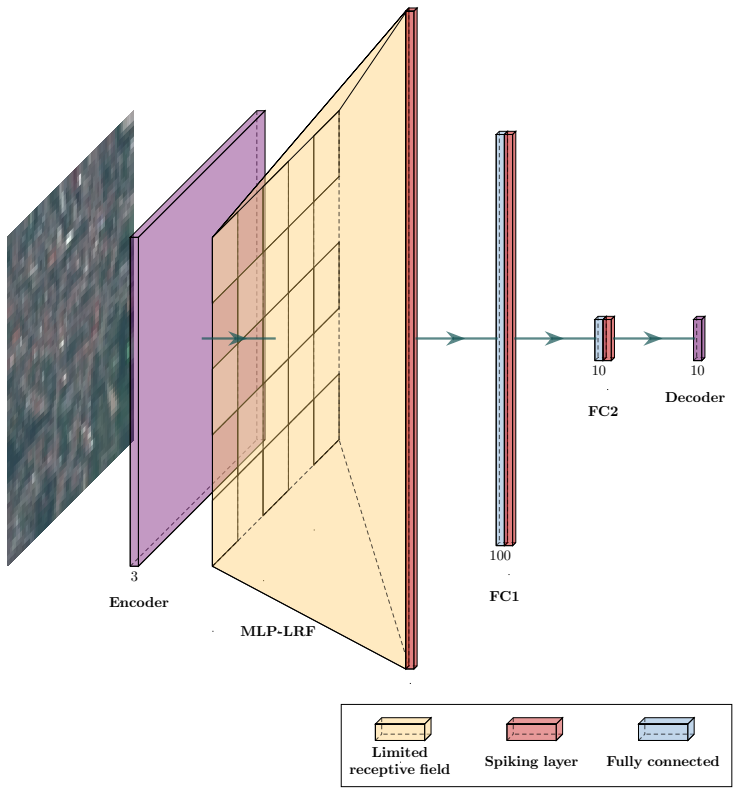


Figure 14: Example of MLP model with LRF layer. Input is divided in independent limited receptive fields, each one mapped to a MLP layer of spiking neurons.

8. Estimation of computational load

One of the aims of this project is the establishment of a more rigorous way to compare energy consumption among different Spiking Neural Networks, and between SNNs and their ANN/DNN counterparts. The final objective is not to develop a method to estimate the absolute energy required, for the number of the unknown parameters is too high for such a task, but rather to achieve a credible way to perform relative comparisons.

When dealing with ANNs in traditional computing hardware (CPU, GPU, TPU), data movement dominates energy consumption at inference, reaching up to 90 % of the total for certain architectures [52, 53]. Hence, even if widely used, number of MAC (FLOP, MVM) and the number of weights are not ideal metrics for energy computation. Hence, energy consumption can be expressed by the sum of two terms:

$$E = E_{\text{comp}} + E_{\text{mem}} \tag{8.1}$$

where E_{comp} is the computational energy, and E_{mem} is the memory energy, consumed to move data (weights and feature maps) across the memory. While the computational energy is directly dependent on the actual number and type of floating point operations, the estimation of the memory energy is more complex, for modern hardware organizes memory in hierarchy with different speed of access. The way in which data flows across the hierarchy is one of the most prominent factors that affect and differentiate the various architectures (CPU, GPU with and without dedicated tensor operations, e.g. Nvidia tensor cores). An accurate estimation of the energy would require the modeling of the data flow, tailored for the specific network and layer architecture (i.e. layer types, dimensions, and sequence) [52]. It would seem reasonable to adopt a black-box approach like in [54] to estimate a general metrics, expressed in GMAC/W (billions of MAC operation per Watt) as an attempt to take into account the average contributions of both computational and memory energy. While such method could be simplistic for absolute comparison between different networks (e.g. a CNN w.r.t. a MLP, which require radically different handling of memory), it could be a viable way to compare the same network architecture on different computing hardware. Nevertheless, this approach comes with its own limitations. In recent years, machine learning-tailored hardware has seen an impressive increment in both absolute performance and power efficiency (see in example [55] compared to [56]), mostly due to specialized computing units (tensor cores, TPUs). But such performances are achieved by means of extreme parallelism, which entails the processing of data in *batches*. In fact, there is a huge difference in energy consumption between batched and non-batched data in such type of hardware, especially due to the fact that most of the memory optimization strategies are way less effective without massive parallelization [57]. Unfortunately, the estimation of the energy spent in non-batched data processing cannot be generalized and would require direct measurement on specific use cases. Moreover, the energy efficiency drops quickly whenever the HW capability is not saturated, due to relatively high power consumption in idle mode, in which a modern GPU can require up to 80 W[4].

The most used metrics to compare SNNs performances are the number of emitted spikes and the number of synaptic operations [26, 46]. Even if a certain increasing trend in energy can expected with increasing number of spikes, different internal connectivity can lead to very different numbers of synaptic operations, making such metric only a very rough estimation. [26] estimates and compare the power consumption of different coding schemes in SNNs by implementing very simple benchmark networks on an FPGA. Useful to compare different coding schemes, this method is hardly effective in comparing large networks and different neuron models, for an equivalent HW implementation should be realized for each architecture. Moreover, a comparison with equivalent ANN models would be impossible.

[4] Measurement taken on a Nvidia RTX A6000 GPU board, mounted in a Intel(R) Xeon(R) W-2275 CPU, 128 GB RAM system. The reported power value is related only to the GPU.

8.1. Assumptions

The aim of this work is to find a *hardware-agnostic* method, capable to put in place a relative comparison of the computational load of SNNs based on different neuron models, and to compare them with their ANN counterparts. To this aim, the number of *Equivalent Multiply and Accumulate Operations (EMAC)* is proposed as metric. Such metric is indeed not suitable to estimate the absolute energy consumption at inference, for several assumptions and experimental measurements related to actual hardware architecture and implementation would be needed; nevertheless, is a good starting point to perform initial trade-offs and relative comparison between network architectures.

> **Remark.** *For sake of simplicity from here forward in this document the terms* energy consumption *and* computational load *of a network are both referred, even if in not completely appropriate way, to the value of EMAC required by the network to perform a single inference.*

Also, the assumptions required for the implementation of the proposed method are minimized:

Relative weights of different types of floating point operations. All the floating point operations performed in standard ANNs are of the Multiply and Accumulate (MAC) type. Conversely, in SNNs a significant part of the computation entails simpler Accumulate (AC) operations, thanks to the binary nature of spikes. Then, different relative weights need to be assigned to AC and MAC in the estimation of the related computational burden. [53] reports a 0.9 pJ energy consumption for AC, 4.6 pJ for MAC, for FP32 number format in a 45 nm CPU architecture, taking into account only the actual computation energy (excluding the memory contribution). In this work, it is assumed that memory movement is the dominant factor in the determination of the energy performance: assuming a unitary weight for MAC, this leads to a conservative value of 2/3 assigned to the AC weight, due to the fact that in an Accumulate operation only 2 numbers are involved, instead of 3 as in the MAC.

Optimal network implementation. No assumption is made on the actual implementation of the networks. Only the strict number of floating point operations necessary for inference is included in the estimation, with no overhead related to input/output operations, system idle power consumption, or software libraries implementation.

Digital implementation All the assumptions above imply a further, underlying assumption, that is the network is implemented in a digital fashion. Although analog neuromorphic processors promise unprecedented improvements in energy consumption [58, 59, 60], a rigorous comparison between different networks would be even harder, being the actual energy consumption extremely dependent on the specific technological solution, often tailored to specific types of network (i.e. neuron models). The assumption of a digital implementation allows the achievement of more reliable comparison between different architectures. Moreover, SNN performances obtained with this method can be considered a conservative estimate, especially in the comparison with ANNs.

8.2. Neuron models

In this section, the number and types of floating point operations required by the different neuron models adopted in this work are identified. In a Spiking Neural Network, part of the operations is performed whenever a spike is received or emitted (*synaptic operations*, while other operations are executed at each time step during the update of the neurons internal states. Then, the total number of equivalent MAC operations $\mathrm{E_{tot}}$ depends on two distinct contributions:

$$\mathrm{E_{tot}} = \mathrm{E_{syn}} + \mathrm{E_{upd}} \tag{8.2}$$

where $\mathrm{E}_{\mathrm{syn}}$ is the synaptic operation contribution, and $\mathrm{E}_{\mathrm{upd}}$ the neuron update. The two term can be further expanded in:

$$\mathrm{E}_{\mathrm{tot}} = s e_{\mathrm{syn}} + nT e_{\mathrm{upd}} \tag{8.3}$$

where s is the total number of synaptic operations performed during the inference, n is the number of neurons in the network, and T is the number of time steps. The two terms e_{syn} and e_{upd} are respectively the energy per synaptic operation and the energy per neuron update and they depend on the specific neuron model. In the following, these two parameters are derived for the different types on neuron considered in this work, expressed in terms of EMAC (as defined in Sec. 8.1). The distinction is still convenient whenever applied to actual energy estimation, for in neuromorphic hardware the single floating point operation can have different energy costs but energy consumption can still be distinguished between synaptic operations and neuron updates.

8.2.1. IFL neuron

The discrete dynamic of an IFL neuron in the Norse framework is described by the following equations (the index of the neuron is omitted for clarity):

$$\begin{cases} i_{k+1} = i_k + \sum_{j=1}^{n_S} w_j S_{jk} + b \\ v_{k+\frac{1}{2}} = v_k + i_{k+1}\Delta t \\ v_{k+1} = v_{k+\frac{1}{2}} - v_{\mathrm{th}} S_{k+1} \end{cases} \tag{8.4}$$

where i and v are the neuron current and potential, respectively; k and $k+1$ identify two subsequent discrete time steps, n_S is the number of pre-synaptic neurons (the neurons in the receptive field of the unit), S_j the received pre-synaptic spikes (the spikes emitted by the jth pre-synaptic neuron), w_j the corresponding weights , b the bias, $S_{k+1} = H(v_{k+\frac{1}{2}} - v_{\mathrm{th}})$ the emitted spikes, Δt the width of the time step. No index is indicated for the bias: in fact in a linear, fully connected layer, there are just one bias for the whole layer, while for a convolutional layer there is a bias for each channel. Nevertheless, from the perspective of the single neuron is irrelevant. Looking to an energy-efficient implementation, the current update can be rewritten as:

$$i_{k+1}\Delta t = i_k \Delta t + \sum_{j=1}^{n_S} w_j \Delta t S_{jk} + b\Delta t \tag{8.5}$$

and since Δt is constant, $i\Delta t = \tilde{i}$, $w_j \Delta t = \tilde{w}_j$, and $b\Delta t = \tilde{b}$ can be treated as single quantities (avoid in this way the multiplication by Δt), leading to:

$$\begin{cases} \tilde{i}_{k+1} = \tilde{i}_k + \sum_{j=1}^{n_S} \tilde{w}_j S_{jk} + \tilde{b} \\ v_{k+\frac{1}{2}} = v_k + \tilde{i}_{k+1} \\ v_{k+1} = v_{k+\frac{1}{2}} - v_{\mathrm{th}} S_{k+1} \end{cases} \tag{8.6}$$

From the point of view of the necessary floating point operations, the contribution of each term in the system (8.6) is:

- accumulation of input spikes: given that the input spikes S_{jk} are binary spikes, $+\sum_{j=1}^{n_S} \tilde{w}_j S_{jk}$ is a **AC** operation every time a spike is received (synaptic operation);
- current bias update: $+\tilde{b}$ is a **AC** op. every time instant (contribution to neuron update);
- potential update, input current contribution: $+\tilde{i}$ is a **AC** op. every time instant (contribution to neuron update);

- potential reset at spike emission: $-v_{\text{th}}S_{k+1}$ is formally an **AC** op. for each emitted spike, but the potential reset can be implemented in different ways, among which the simplest is just the assignment of $v = v_{\text{reset}}$ that is not even a floating point operation. *The contribution of spike emission to total energy consumption is neglected.*

The energy parameters for the IFL neuron can be then expressed as:

$$\text{IFL:} \begin{cases} e_{\text{syn}} = 1\,\text{AC} = 0.667\,\text{EMAC} \\ e_{\text{upd}} = 2\,\text{AC} = 1.333\,\text{EMAC} \end{cases} \tag{8.7}$$

8.2.2. LIF neuron

The discrete LIF dynamics in Norse can be described as:

$$\begin{cases} i_{k+1} = i_k - i_k \dfrac{\Delta t}{\tau_{\text{syn}}} + \sum_{j=1}^{n_S} w_j S_{jk} + b \\ v_{k+\frac{1}{2}} = v_k + (i_{k+1} - v_k)\dfrac{\Delta t}{\tau_{\text{mem}}} \\ v_{k+1} = v_{k+\frac{1}{2}} - v_{\text{th}} S_{k+1} \end{cases} \tag{8.8}$$

Following the same procedure as seen for the IFL neuron, the contributions to the energy parameters are:

- accumulation of input spikes: $+\sum_{j=1}^{n_S} w_j \Delta t S_{jk}$ is a **AC** operation every time a spike is received (synaptic operation);
- current update, exponential decay: $-i\dfrac{\Delta t}{\tau_{\text{syn}}}$ is a **MAC** operation every time instant (contribution to neuron update). Please note that it is no more possible here to aggregate $i\Delta t$ as in the IFL case;
- current update, bias: $+b$ is a **AC** op. every time instant (contribution to neuron update);
- potential update: $+(i - v)\dfrac{\Delta t}{\tau_{\text{mem}}}$ is an **AC** op. (the parenthesis), followed by a **MAC** op. every time instant (contribution to neuron update);
- potential reset at spike emission: this contribution is neglected (see IFL neuron at Sec. 8.2.1).

Then, the energy parameters for the LIF neuron are:

$$\text{LIF:} \begin{cases} e_{\text{syn}} = 1\,\text{AC} = 0.667\,\text{EMAC} \\ e_{\text{upd}} = 2\,\text{AC} + 2\,\text{MAC} = 3.333\,\text{EMAC} \end{cases} \tag{8.9}$$

8.2.3. IF (Mostafa 2017) neuron

The discrete dynamics of this neuron can be described as follows:

$$\begin{cases} i_{k+1} = i_k - i_k \dfrac{\Delta t}{\tau_{\text{syn}}} + \sum_{j=1}^{n_S} w_j S_j \\ v_{k+\frac{1}{2}} = v_k + i_{k+1}\Delta t \\ v_{k+1} = v_{k+\frac{1}{2}} - v_{\text{th}} S_{k+1} \end{cases} \tag{8.10}$$

As in the IFL model, some operations can be aggregated to optimize the computation. The current update can be rewritten as:

$$i_{k+1}\Delta t = i_k \Delta t - i_k \Delta t \frac{\Delta t}{\tau_{\text{syn}}} + \sum_{j=1}^{n_S} w_j \Delta t S_j \tag{8.11}$$

again, $i\Delta t = \tilde{i}$ and $w_j \Delta t = \tilde{w}_j$, leading to the simplified system:

$$\begin{cases} \tilde{i}_{k+1} = \tilde{i}_k - \tilde{i}_k \dfrac{\Delta t}{\tau_{\text{syn}}} + \sum_{j=1}^{n_S} \tilde{w}_j S_j \\ v_{k+\frac{1}{2}} = v_k + \tilde{i}_{k+1} \\ v_{k+1} = v_{k+\frac{1}{2}} - v_{\text{th}} S_{k+1} \end{cases} \tag{8.12}$$

The energy parameters can be now obtained from the system (8.12):

- accumulation of input spikes: given that the input spikes S_{jk} are binary spikes, $+\sum_{j=1}^{n_S} \tilde{w}_j S_{jk}$ is a **AC** operation every time a spike is received (synaptic operation);
- current update, exponential decay: $-\tilde{i}_k \dfrac{\Delta t}{\tau_{\text{syn}}}$ is a **MAC** op. every time instant (contribution to neuron update);
- potential update, input current contribution: $+\tilde{i}$ is a **AC** op. every time instant (contribution to neuron update);
- potential reset at spike emission: this contribution is neglected (see IFL neuron at Sec. 8.2.1)

Then, the energy parameters for the IF (Mostafa 2017])neuron are:

$$\text{IF (Mostafa 2017):} \begin{cases} e_{\text{syn}} = 1\,\text{AC} = 0.667\,\text{EMAC} \\ e_{\text{upd}} = 1\,\text{AC} + 1\,\text{MAC} = 1.667\,\text{EMAC} \end{cases} \tag{8.13}$$

8.3. Estimation procedure

The energy consumption at inference is then computed by means of Eq. (8.3), here reported for clarity:

$$\text{E}_{\text{tot}} = s e_{\text{syn}} + n T e_{\text{upd}} \tag{8.14}$$

Given the specific network architecture, the number of neurons n and the number of time steps T are immediately known, while a way to estimate the number of synaptic operations s is needed. The computation can be performed layer-wise. The first contribution to the total synaptic operations derives by the flowing of spikes between subsequent layers. The number of synaptic operations performed between a layer l and the previous layer $l-1$ can be estimated by:

$$s_{(l)} = n_{s(l)}\, n_{n(l)}\, T\, P_{(l-1)}(S) \tag{8.15}$$

where n_{s} is the number of pre-synaptic connections in the neuron receptive field, and n_{n} is the number of neuron in the layer, both dependent on the type of layer (convolutional, fully connected etc.). T is the number of time steps, and $P_{(l-1)}(S)$ is the probability that a pre-synaptic neuron emits a spike. This last term can be estimated by:

$$P_{(l-1)}(S) \sim \frac{N_{s(l-1)}}{T\, n_{n(l-1)}} \tag{8.16}$$

in which $N_{s(l-1)}$ is the total number of spikes emitted by the previous layer at inference. Substituting (8.16) in (8.15) we obtain:

$$s_{(l)} \sim n_{s(l)}\, n_{n(l)}\, T\, \frac{N_{s(l-1)}}{T\, n_{n(l-1)}} = n_{s(l)}\, n_{n(l)}\, \frac{N_{s(l-1)}}{n_{n(l-1)}} = n_{s(l)}\, n_{n(l)}\, f_{(l-1)} \tag{8.17}$$

where f_{l-1} is the spiking rate *of the previous layer.*

Additional synaptic operations are performed inside *recurrent layers*. In this case, the layer output spikes are routed recursively as input to the layer itself: depending on the type of layer (convolutional, fully connected, etc.), such recursive routing is itself a convolution or a linear connection. In this case, the number of synaptic operations due to the recursion $s_{r(l)}$ can be computed as:

$$s_{r(l)} = n_{s(l)}\, n_{n(l)}\, f_{(l)} \tag{8.18}$$

which is similar to Eq. (8.17) except that the spiking rate *of the same layer* $f_{(l)}$ is used.

TTFS decoding. In case of Time-To-First-Spike and rank order encoding, the inference ends de facto at the time the first spike is emitted by the output layer: that time instant shall be considered for estimation of spike rate and number of time steps, leading to an even lower computational burden. This implies also that the inference time T is not fixed and can vary for each single inference: latency performance can be still assessed in a statistical way. Conversely, for rate-based encoding the time of inference T is constant and predefined.

8.3.1. Computation of connectivity parameters

In this section the computation of the connectivity parameters n_s and n_n is expounded for different layer types.

2D Convolution. For a `Conv2d` layer:

$$n_{\mathrm{s}} = k^2 C_{\mathrm{in}} \tag{8.19}$$

$$n_{\mathrm{n}} = whC_{\mathrm{out}} \tag{8.20}$$

where k is the kernel size, w and h are the two spatial dimensions of the *output* (width and height after the convolution), C_{in} and C_{out} the number of respectively input and output channels. The output dimensions w and h can be computed with the parameters of the convolutional layer with the standard formula:

$$h = \mathrm{floor}\left(\frac{h^- + 2p - k}{s}\right) + 1 \tag{8.21}$$

where h^- is the height before the convolution, p is the padding, k the kernel size, s the stride. The same formula is used also for the computation of the width w.

Fully connected layer. The computation of connectivity parameters for a `Linear`, fully connected layer is simply:

$$n_{\mathrm{s}} = C_{\mathrm{in}} \tag{8.22}$$

$$n_{\mathrm{n}} = C_{\mathrm{out}} \tag{8.23}$$

MLP with limited receptive field. A perceptron layer with limited receptive field is equivalent to a convolution with a single output channel ($C_{\mathrm{out}} = 1$) and kernel size (which is the size of the receptive field) equal to the stride value ($k = s$). Equations (8.19), (8.20), and (8.21) are used, with a proper selection of parameters.

8.3.2. Equivalent ANN

For a ANN, floating point operations happen for all the inputs of each neurons, but without the time dimension and the states updates. So for each layer we have just one contribution:

$$n_{\mathrm{S}}\, n_{\mathrm{N}}\, E_{\mathrm{MAC}} \tag{8.24}$$

The nonlinear output function of the ANN neurons can or cannot contribute to the energy consumption: i.e. a ReLU is in practice not a floating point operation at all, and it could be neglected; more complex functions, like Sigmoid Tangent or SeLU, involve some computation and could have an impact. *For simplicity, we stick to ReLU and do not consider any additional contribution by output function.*

8.4. Known limitations

The energy estimation approach here presented has both strengths and weaknesses. The main advantage is the achievement of a metric more accurate than the mere number of spikes (see Sec. 9.3), capable to compare very different models (different encoding, different neurons models, and even SNNs w.r.t. ANNs). In analyzing the results, it must be taken into account that some network architectures are not equally efficient while implemented on different hardware. In fact, while ANN are nowadays extremely efficient on hardware like GPU and TPU (especially if batched computation is applicable), their implementation on neuromorphic hardware could be even impossible. The opposite is valid for SNNs: while on neuromorphic devices they promise to be extremely efficient in terms of energy, with no need to operate in batch, their implementation on standard hardware is complex, if efficiency is desired: the use of libraries such PyTorch make prototyping and training very easy, but slow and memory consuming, for the network is to be unrolled in time. Moreover, while the network is trained offline it still could be possible to process input in batches, but in a real world application this could be easily not possible. In this, SNNs on neuromorphic hardware would have a clear advantage. Another issue that makes the estimation of absolute energy even more hardware-dependent is the fact that on standard computing devices (CPU/GPU/TPU) the relative weight of a floating point operation is always the same, while it is not necessarily the same on neuromorphic hardware, where, even in digital implementation, synaptic operations and neuron update may happen in different parts of the processor, leading to different energy consumption even for operations that formally are of the same type [51].

Still, the approach followed in this work maintain a certain usefulness. Being hardware-agnostic, it allows relative comparisons between architectures, enabling at least preliminary information for trade-off. Moreover, in case a SNN would achieve a better adimensional energy consumption w.r.t. its ANN counterpart, it can be expected that such advantage would be even larger once ported on neuromorphic hardware.

In the current approach, there are some known factors that lead to a potential overestimation of the adimensional energy. In absence of biases, the internal state (post-synaptic current and potential) of each neuron does not require updates until the first input spike is received. As the number of layer increases, this factor becomes more relevant. Moreover, when neuron that spike once at most are used, after a spike is emitted the neuron goes silent and its update can be stopped. In any case, such kind of optimization could be not necessarily available in actual implementation, depending on the hardware and/or the API used. For this reason they were not included in the energy estimation process.

There are also underestimation factors. In fact, the adopted approach does not take into account any energy consumption contribution but the operations strictly necessary for the inference. In particular, idle power consumption and possible overhead due to general API implementation are not taken into account: these two terms can have a very large variability depending on the actual hardware platform. Their general effect would be to reduce relative differences.

9. Numerical results

In this section, the tests carried out to assess the potential performance of SNNs are expounded, and the achieved results are discussed. Several architectures have been trained, SNNs and their ANN counterparts, to compare their performance in terms of accuracy vs EMAC.

9.1. Test cases

The 57 test cases run in this activity are summarized in Table 3 in Appendix A. If not otherwise specified, all the cases are trained with the Superspike surrogate gradient method [31, 18] in a customized Norse framework [61]. Test cases belong to four main categories:

- spiking Multilayer Perceptrons, IFLOnce neurons, latency encoding;
- spiking Multilayer Perceptrons with Limited Receptive Field, LIF neurons, rate encoding;
- spiking Convolutional Neural Networks, IFLOnce neurons, latency encoding;
- spiking Convolutional Neural Networks, LIF neurons, rate encoding.

Also, their corresponding ANN counterparts have been trained to have a term of comparison. All the ANNs adopt ReLU as activation function:

- Multilayer Perceptrons;
- Multilayer Perceptrons with limited receptive field;
- Convolutional Neural Networks.

Moreover, some special cases have been trained to test specific features:

- *P020* tests the effectiveness of LIF neurons with latency encoding and SG training;
- *dodo_a* is a convolutional network with LIF network and a different architecture w.r.t. the VGG-style used as general benchmark (see Sec. 7.2);
- *dodo_b* is a MLP with IF neurons (one spike at most) with latency encoding, trained with optimization of spike times like in [47], to compare this training method with SG;
- *dodo_c* is a recurrent MLP with LIF neurons and rate encoding, designed to assess the effect of recurrence in spiking layers.

In the following sections, the achieved results are shown and their implications are discussed.

9.2. Accuracy vs EMACS

The first and most important figure of merit is the achieved accuracy w.r.t. the energy consumption. Figure 15 shows the obtained results. Not all the test cases are shown, but just the ones which attain the best performance for each of the groups mentioned in Sec. 9.1, to highlight what could be considered a Pareto front of the achieved performance. The box at the bottom right shows the distribution of the test cases among the aforementioned groups.

The first, and most notable, result is that SNNs are able to reach performance comparable of ANNs, with significantly less floating point operations. This is particularly evident looking at the latency architectures *P032*, *P036*, and *P039*, corresponding respectively to the ANNs cases *PANN16*, *PANN13*, and *PANN14*: a $\sim 2.5\,\%$ drop in efficiency corresponds to a $\sim 60\,\%$ decrease in the number of floating point operations per inference. Performances and not equally good for rate based networks (*P041*, *P021*, *P019*) with the benchmark architecture (Sec. 7.2), which have a slightly worse accuracy w.r.t. latency ones, coupled with an energy consumption worse than the ANNs counterparts. But the

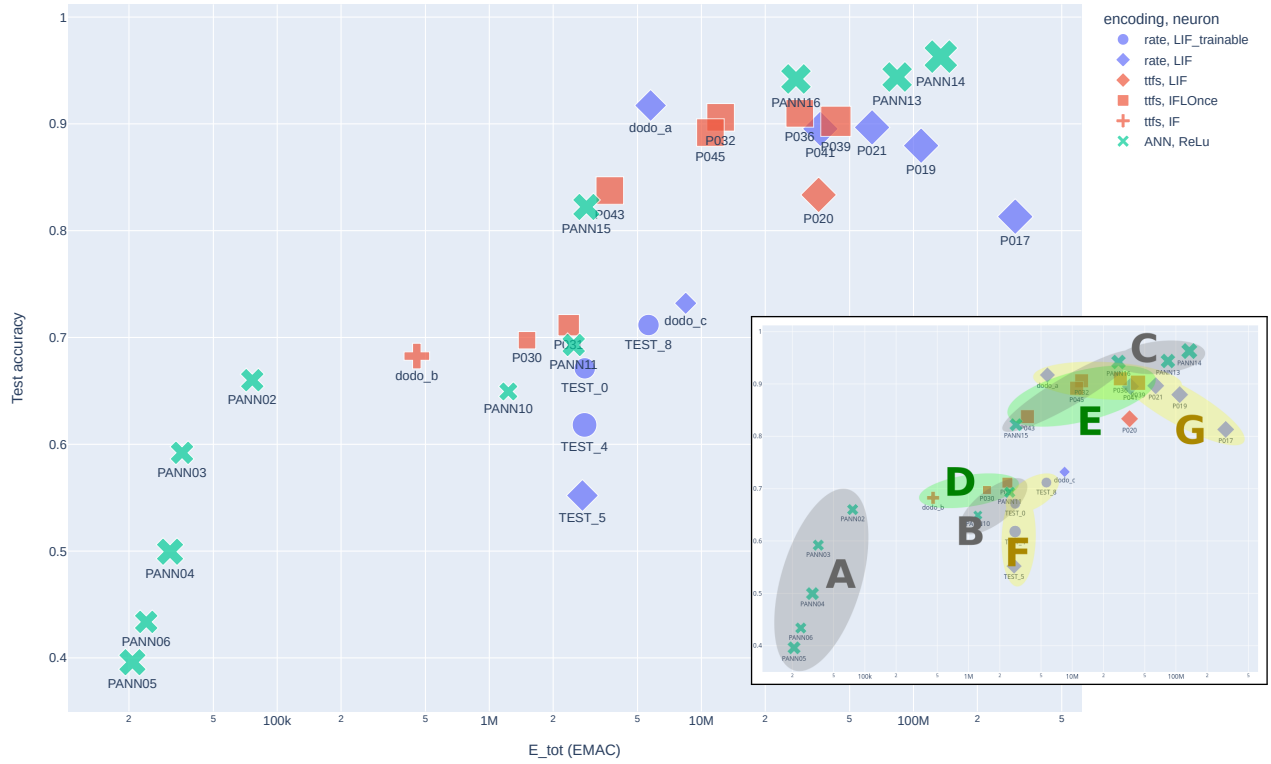


Figure 15: Accuracy vs adimensional energy (EMAC). The dimension of the marker is proportional to the number of the network layers, as a rough index of the network complexity. In the square at the lower right, the main groups of test cases are higlighted: A) ANN – MLP with limited receptive field; B) ANN – MLP; C) ANN – CNN; D) SNN – MLP with TTFS encoding, IF neuron; E) SNN – CNN with TTFS encoding, IF neuron; F) SNN – MLP with rate-based coding, LIF neuron; G) SNN – CNN with rate-based coding and LIF neuron. A part of the main groups, *dodo_c* is a SNN MLP with recurrence and LIF neuron, while *P020* is a convolutional SNN with TTFS encoding and LIF neuron.

alternate architecture *dodo_a* hits better performance than latency ones on both accuracy and EMAC. This suggests that architectures that are good for standard ANNs could not necessarily be the best ones for Spiking Neural Networks.

The opposite can be observed for simpler networks: both MLPs and MLP with Limited Receptive Field present increased performance w.r.t. their ANNs counterpart in term of accuracy, highlighting the potential of spiking system to store a large amount of information efficiently. Nevertheless, for such simple networks, this is achieved at the expense of a larger amount of floating point operations (EMAC). Moreover, the accuracy of spiking MLP and MLP-LRF networks still remains relatively low, under 75 %, a value uncompetitive with the CNNs and the SCNNs, albeit better than their ANNs counterparts. It must be said that a prominent role in the determination of the total EMAC per inference in SNNs is played by the number of time steps in the simulation, a hyperparameter whose optimization has not been specifically pursued in the this work. Its role is even more important in simple networks in which the reduced number of neurons forces the information to be compressed, reducing in this way the sparsity of the computation (with a high number of synaptic operations due to the fully connected architecture of the MLPs). In this case, a non optimized number of time steps acts as a multiplier in the estimation of the EMAC per inference. The limited receptive field, applicable only to the first layer in the network to cope with this specific problem, seems not sufficient to significantly limit the impact of the subsequent fully connected layers. Conversely, the inherently small receptive field of the convolutional layers is particularly apt to exploit the sparsity of the spiking computation. A more detailed analysis on the different sources of energy consumption inside the networks at inference time is included in Sec. 9.3.

Case *P020* shows how the coupling of LIF neurons with latency encoding and SG training suffers a loss in accuracy.

The effect of recurrence has been analyzed in case *dodo_c*: despite a certain increase in accuracy in a very compact model, being the recurrence a way to store additional information in the same layer, the number of EMAC is significantly increased as well. Being the recurrent layer placed at the bottom of the layer stack, it happens in a relatively large layer, with a significant impact over the number of synaptic operations. The impact of recurrence on higher and/or convolutional layers is still an open point. Additional considerations about the impact of the internal network connectivity to the total number of EMACs per inference are made in Sec. 9.3.

Case *dodo_b* has been adopted to compare latency encoding trained with SG method to a similar network trained with Backpropagation Through Time (BPTT) in the flavor described in [36]. The training achieved a remarkably low level of EMAC per inference, compared to similar networks trained with SG. Nevertheless, BPTT showed to require a huge computational effort in the training phase. To achieve a functional network on the available computational resources, the input required to be scaled to 16×16 pixel. Such scaling has for sure contributed to the reduction in energy consumption but also poses doubts to possible increase in performance, involving a loss of information. Moreover, the huge computational effort could make the scaling to larger models impracticable. With respect to Surrogate Gradient, since spike times are directly computed, there is no way to impose an upper bound on the maximum simulation time, and since spike times are computed analytically, there's no pre-determined time step, with possible issues for a digital implementation on hardware. To perform a proper comparison, an arbitrary selection of $\Delta t = 1\,\text{ms}$ has been made. For an optimized application, a proper Δt selection is required, in order to minimize the number of time steps at inference, preserving the necessary amount of time resolution to avoid degradation in information decoding. All these issues make the BNTT option less viable with respect SG training.

9.3. EMAC vs number of spikes

In this section the metric here proposed as harware-agnostic proxy of the energy consumption at inference (EMAC) is compared with the most simple estimation often adopted in literature as first estimation, that is the number of emitted spikes at inference.

Figure 16 shows the distribution of the two quantities across the test cases listed in Table 3 in Appendix A. The average values evaluated on the test set are shown. If a general trend is present (as the spikes increase, the EMACs are generally larger) it can be seen how limiting the estimation to the spikes only could lead to large errors with a large variation of computational load among cases with almost the same number of spikes, and vice versa. Moreover, this happens the most in the region of maximum interest, the one at low energy/EMAC, zoomed in the box at lower-right in the Figure.

The reason of this behavior reside in the fact that a significant part of the EMAC per inference is due to synaptic operations. Hence, the actual routing of spikes between layers, and the network's internal connectivity pattern can change drastically the overall EMAC even with a similar number of emitted spikes. Figure 17 shows the breakdown of EMAC per inference among neuron update, synaptic operations, and recurrent synaptic operations for some selected test cases, while Fig. 18 shows their respective number of emitted spikes. Histograms show the average value, while error bars shows the standard deviation. Cases *P004* and *P005* share the same convolutional architecture, in terms of neuron types, latency encoder/decoder, and number and size of spiking layers. The only difference between the twos is the connection pattern between convolutional layers: in *P004* convolution is performed with kernel size $k = 3$ and stride $s = 1$, and dimensional reduction is achieved with standard maxpool layers; conversely, in *P005* convolution is performed with $k = 5$ and $s = 2$, with no maxpooling. Due to the larger receptive field of neurons, *P005* exhibits a $\sim 45\,\%$ increase in EMAC/inference, even with $\sim 10\,\%$ less spikes. Looking at the EMAC breakdown, sharing the same number and type of neurons, and with an almost equal average number of time steps at inference (38.1 ms and 37.1 ms) *P004* and *P005* spend a practically equal amount of EMAC for neurons update: the totality of the increase is instead due to a large difference in the energy spent for synaptic

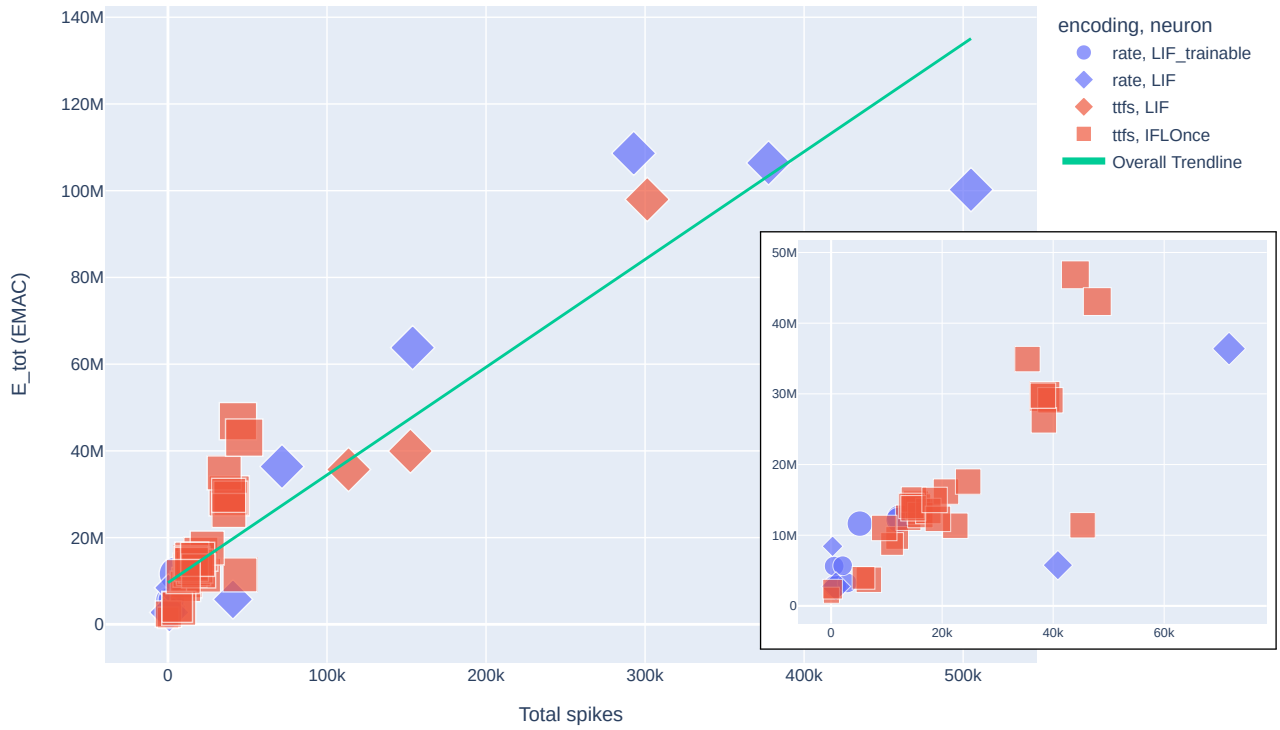


Figure 16: Adimensional energy (EMAC) w.r.t. total emitted spikes per inference (average values over the test set).

operations. From the energy consumption perspective, it would appear advisable to privilege a low level of internal connectivity (also takiong into account consideration made in Sec. 9.2). However, it must be said that a larger receptive field in the convolution seems to bring significant benefit to the system performance, with *P005* scoring a 5.3 % in the absolute test accuracy. A careful trade-off between energy and accuracy should be performed in the determination of am effective architecture.

The discrepancy between the number of spike and the computational load is even more evident in the case *dodo_a*, still a convolutional architecture. Being a rate-based network, the number of emitted spikes is high, and almost doubles the values achieved by *P004* and *P005*. Nevertheless, the lower number of time steps (32 ms), and the lower size and number of layers make the average value of EMAC/inference almost a third w.r.t. *P005*.

P030 and *dodo_c* are another notable example showing the importance of the internal connectivity. They both share the same MLP architecture with one hidden layer of 100 neurons, but while *P030* is a standard MLP, with IFLOnce neurons and latency encoding, the hidden layer of *dodo_c* is made by recurrent LIF neurons with rate encoding. From Fig. 17 is possible to see how the computational effort required at inference due to neuron update is practically the same (lighter IFLOnce neuron are compensated by a higher number of time steps at inference). But recurrence involves a higher number of synaptic operations, leading to a value of EMAC per inference more than 5 times higher. In case *dodo_c*, $\sim 92\,\%$ of the computational effort is due to synaptic operations in the recurrent layer.

9.4. Effectiveness of regularization

In this section, the results observed by the use of the regularization techniques detailed in Sec. 6 are discussed.

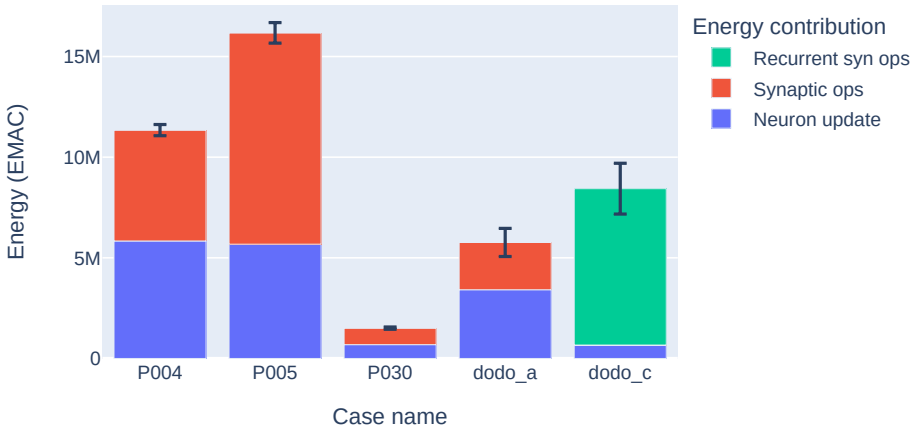


Figure 17: EMAC per inference for selected cases: breakdown among network tasks, average value over the test set. The standard deviation value is indicated by black error bars.

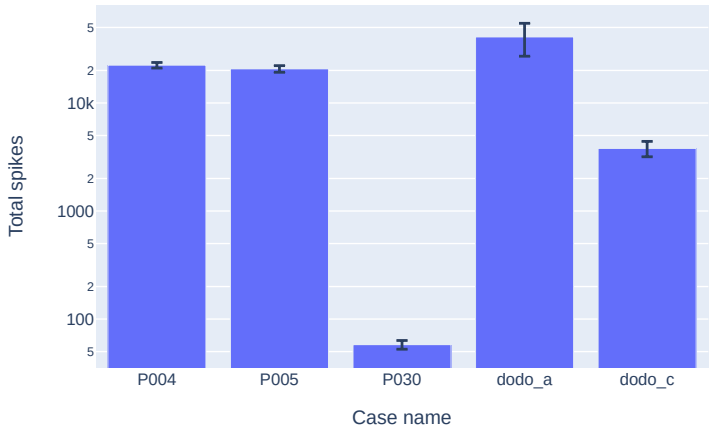


Figure 18: Total emitted spike per inference, average value over the test set. The standard deviation value is indicated by black error bars..

9.4.1. Target output time

In preliminary tests, the use of target output time regularization (see Sec. 6.1) showed only marginal improvements in final test accuracy, when applied in SG training. See as example, the test cases *P006* (with regularization) and *P016*/*P018* (without). Nevertheless, a certain speedup in training has been detected, proving a certain beneficial effect in the gradient flowing induced by the temporal penalty term in Eq. (6.2). Except for some selected cases, this regularization has been used by default in latency-based, SG trained networks.

9.4.2. Sum of synaptic weights

When applied to SG trained networks, the regularization scheme explained in [45] (see Sec. 6.2) did not prove beneficial (see cases *P013* and *P014*), with an even slightly decreased test accuracy value. The scheme adopted by [36] was used in case *dodo_b*, which implements the same network scheme of the original paper.

9.4.3. Batch Normalization Through time

Figure 19 shows the impact of Batch Normalization Thorugh Time (BNTT, see Sec. 6.3). The histograms report test accuracy, EMAC/inference, and number of emitted spikes per inference for three test cases: *P005*, *P016*, and *P032*. These models entails the same convolutional architecture, with IFL neurons spiking once at most, and latency encoding. The only difference is that *P005* entails no BNTT; neuron-wise BNTT is applied in *P016*; spatial BNTT is applied in *P032*. The number of spikes is similar in all the 3 cases (with a lightly reduced value for *P016*). Neuron-wise BNTT seems beneficial for the energy consumption, but with no improvement on the test accuracy. Spatial BNTT improves both the accuracy, with a 5.4 % gain in the absolute test accuracy, and the energy consumption, with a 24 % drop in EMAC per inference w.r.t. case *P005*.

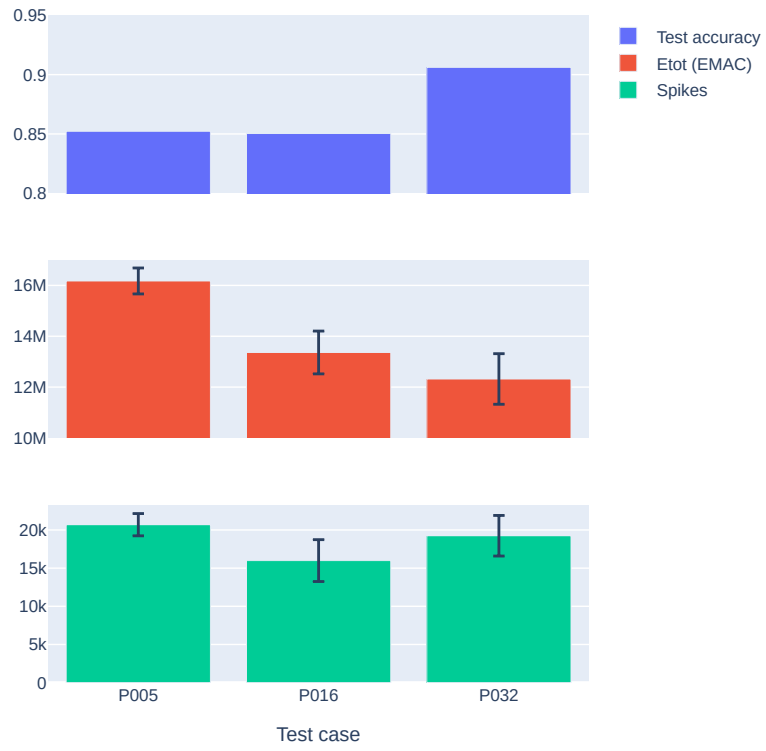


Figure 19: Effect of Batch Normalization Through Time. The three test cases shares the same convolutional architecture presented in Sec. 7.2, except for the application of BNTT. P005) no BNTT; P016) neuron-wise BNTT; P032) spatial BNTT.

Moreover, BNTT can give some insights on the internal functioning of the network. Figure 20 shows the values of the output scale parameter γ for the two convolutional layers in case *P032*. In each layer, there is a value of γ fore each time step, for each convolution channel, since spatial BNTT is applied. In this case, γ are initialized at 1 at the beginning of the training. It is clearly visible ho a temporal pattern is correctly learned for each channel. But in both the layers is possible to see which is the time interval in which the training has changed the value of the parameters: from zero to 25 ms for the first layer, from 3 ms to 40 ms for the second one. This range is shifted forward in time for the higher layer, as it could be expected, as it takes a certain amount of time for the potential in the lower layer to increase enough to spike. But this information shows also which is the time range in which each layer affects the network output, i.e. the fact that backpropagation has left the values of γ in layer 2 unchanged after 40 ms is an indication that any spike emitted by the layer after that time has a null or negligible impact on the final output. This information could be exploited in future developments to further optimize the training process.

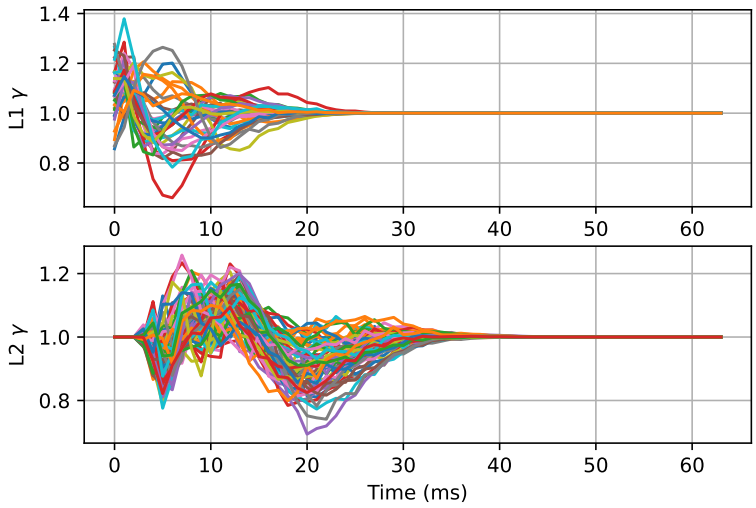


Figure 20: *P032*, Batch Normalization Through Time, learned γ in the two convolutional layers. The network is capable to identify a temporal pattern in the incoming spike trains.

9.5. Scaling to deeper architectures

Despite the regularization schemes tested in this work, and the generally promising performances achieved in accuracy at inference, SNNs still show a certain struggle in scaling to deeper architectures. In this section, the behavior of SNNs with respect to scaling is analyzed. Figure 21, a close-up of Fig. 15, shows the performances, in term of test accuracy vs energy (average EMAC per inference) of the most notable convolutional architectures tested in this work. Differently from Fig. 15, not just the top performing models are shown, but also some other test cases designed to study the performances achieved when equal architectures (in term of layer type and sequence) are applied to different types of network (latency-based SNNs, rate-based SNNs, ANNs). Table 1 summarizes the test cases of Figure 21, showing in a synoptic way the correspondences of architectures, ordered in increasing complexity.

All the latency-based SNNs adopt a ranking order coding, with IFL neurons set to spike once at most, and they are mostly variations of the benchmark architecture detailed in Sec. 7.2. Latency based encoding is achieved by means of a convolutional version of of a constant current IAF (with one spike at most, see Secs. 4.2.2 and 4.3). All latency-based models are trained with BNTT applied to convlutional layers (Sec. 9.4.3). Among the rate-based networks, all the networks adopts a *max_voltage* decoder, while the encoder is different: in *dodo_a*, the encoding is obtained by convoluting the input image, and feeding the convolution output as a constant input to a layer of LIF neurons; *P019* adopts

a convolutional variant of a constant current LIF encoder (see Sec. 4.1.1; in case *P021*, whilst being a rate-based network, the same latency encoder mentioned above for latency-based cases is adopted as a way to limit the number of spikes at inference. Case *P041* is the same model of case *P021*, but the number of time step at inference is truncated at the minimum with no performance degradation, evaluated on the validation set (while performances shown in Fig. 21 are evaluated on the test set). This allows to significantly limit the energy consumption of a rate-based with max voltage decoder, with a minimal impact on the accuracy. The included ANN models are the respective counterparts of the SNNs.

Figure 21: Scalability of SNNs. Spiking Neural Networks still struggle when scaled to deeper architectures, w.r.t. ANNs.

The synoptic table allows us to discern different trends as the architecture complexity increases, among the different network types. It can be seen that convolutional ANNs follows a general trend in which as the network complexity increases, scaling to deeper architectures, also the accuracy tends to improve. Even if at a certain point also the training hyperparameters and the selection of the correct LR scheduling assumes a certain importance (see *PANN09* as example, in which the accuracy drops despite the higher number of layers), it is known that VGG-style ANNs of moderate deepness are capable to achieve state of the art accuracy >99 % [62]. A different behavior is observed for Spiking Neural Networks: while at lower complexities they even outperform ANNs, showing the potential of spiking neuron in storing information, as the network deepness increases they reach a sort of maximum: at this point they seems to preserve a good performance, with a clear advantage in terms of energy (as EMAC per inference), w.r.t. their ANN counterparts. But as the complexity increases, the performance starts to drop to unsatisfactory levels of accuracy.

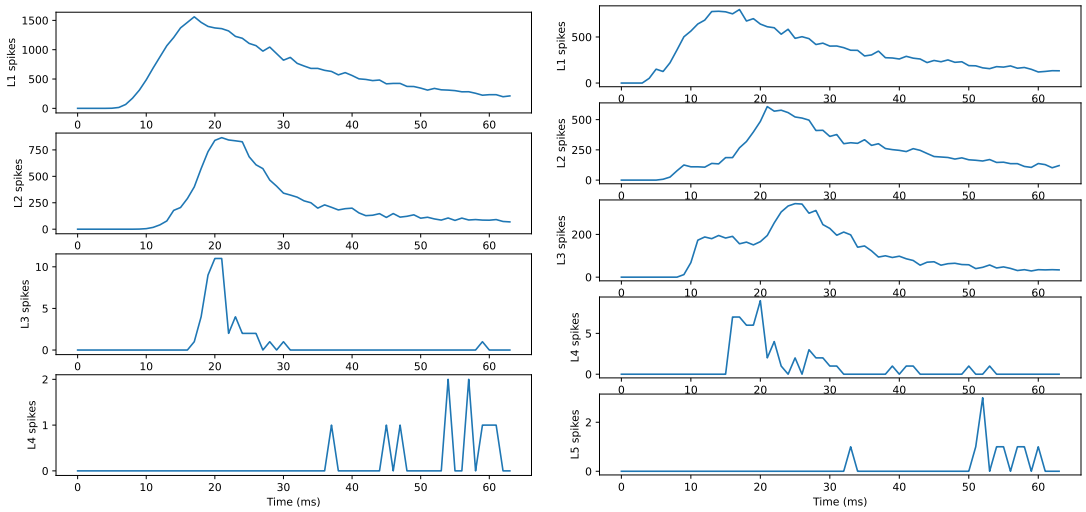
A deeper insight of this phenomenon can be obtained by looking at the internal dynamics of the networks. Figure 22 shows the temporal trend of the number of spikes emitted by each layer of different models with increasing number of layers. All the models in the figure are latency based: *P036*, *P039*, and *P033* are also included in Fig. 21; *P012* is a 11-layer model (10 plus the convolutional spiking encoder) of the same network type that failed training (with a final test accuracy equal to 14.26 %, slightly higher w.r.t. a random guess). *P036* in Fig. 22a represents a case of well trained SNN: in the two convolutional levels, the layer takes a certain amount of time to collect spikes from its

Table 1: Synoptic table with test cases in Fig. 21. Cases are ordered in increasing complexity and deepness of the network. Convolutional encoder, if present, is included in the layers listed in the "Architecture" field.

Input size	**Kernel, stride**	**Architecture**	**Latency**	**Rate**	**ANN**
32	k3, s1	C(20), MP(2), C(40), MP(2), C(20),FC(10)	P044	dodo_a	PANN15
32	k5, s2	C(16), C(32), C(64), FC(100), FC(10)	P043		
32	k5, s2	C(32), C(64), C(128), FC(100), FC(10)	P045		PANN16
64	k5, s2	C(16), C(32), C(64), FC(100), FC(10)	P032	P019, P021, P041	PANN01
64	k5, s2	C(16), C(64), C(128), FC(100), FC(10)	P036		PANN13
64	k5, s2	C(16), C(64), C(128), C(256), FC(100), FC(10)	P039		PANN14
64	k5, s2	C(16), C(32), C(32), C(64), C(64), FC(100), FC(10)	P033		
64	k5, s2	C(16), C(16), C(16), C(32), C(32), C(64), C(64), C(128), C(128), FC(100) ,FC(10)			PANN09

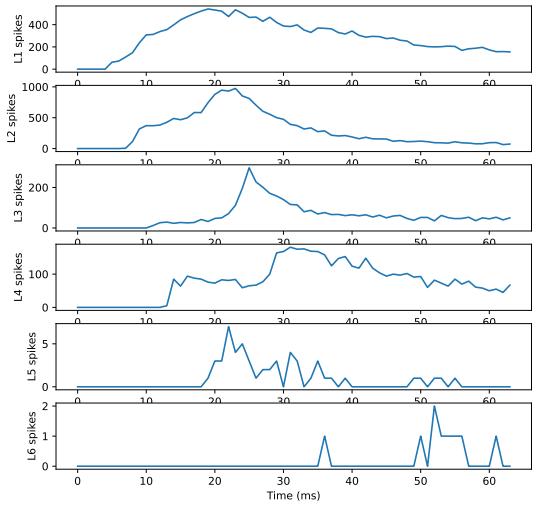
predecessor, and the neurons' potential increases. As the voltage levels approach the threshold, the number of emitted spikes starts to increase, reaches a maximum, and then slowly decrease, as less significant neurons spike. As it can be expected, the response of subsequent layers is slightly translated along time, as each layer needs to accumulate a certain amount of spikes emitted by the layer before. In case *P0339* (Fig. 22b), with just one convolutional layer more, the trained response is already suboptimal. In fact, at the second layer the rise in spikes number proceeds in two steps: first, around 6 ms the spike level rises, to stabilize at ∼9 ms; a second, more relevant rise starts at 15 ms, culminating in the layer's peak at ∼20 ms. At layer 3, this behavior repeats even more evident, with a secondary peak at ∼12 ms and the main peak at 25 ms; this leads the subsequent layer 4 to its peak of activity *well before the previous layer has emitted its most relevant spikes*. The synchronicity is recovered between layers 4 and 5 (the output layer) but that means that the final output *depends on partial information*.

As the number of layers increases, this behavior is even more pronounced. In case *P033* in Fig. 22c, layer 5 reaches its top activity before the peak is reached by the two previous layers, 4 and 3. As the deepness continue to increase, this phenomenon compromises the training. Case *P012* in Fig. 22d is a clear example: spike activity in layer 5 rises and reach its peak in response of a very few input spike from the previous layer. In this way, the most of the information included in the input is not involved in the inference, resulting in a poor final accuracy. More investigation are needed to unlock successful training on deeper networks. Two possible directions for future investigations are possible: network initialization and regularization during training.

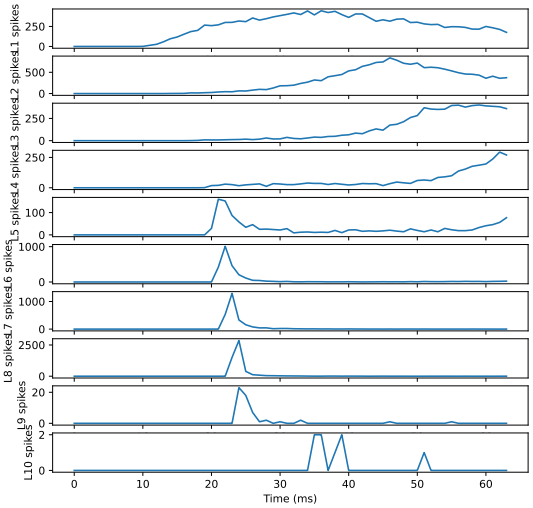


(a) Case P036 – effectively trained network.

(b) Case P039 – suboptimal training.



(c) Case P033 – suboptimal training.



(d) Case P012 – failed training.

Figure 22: Scaling to deeper networks with SCNN and rank order coding (IFLOnce neurons). In the 3 examples above, batch normalization through time is applied on convolutional layers (all except the last two). As the number of layers grows, subsequent layers fail to collect most of the information of lower layers, and starts to emit spikes basing only on a few presynaptic spikes.

10. Conclusion

A preliminary investigation of the potential benefits of Spiking Neural Networks based on temporal coding for onboard Artificial Intelligence applications in space was carried out in this work. As case study, a scene classification task was considered, based on the EuroSAT RGB dataset. SNN models, and their ANN counterparts, were compared in term of accuracy and complexity.

A new metrics, Equivalent MAC operations (EMAC) per inference, was developed to estimate the relative computational load in a hardware-agnostic way. With respect to spikes count or number of synaptic operations, EMAC is suitable to assess the impact of different neuron models, to weight separately contributions given by synaptic operations w.r.t. neuron updates, and to compare SNNs with their ANN counterparts. Albeit the version presented in this study achieves adimensional estimation, and then is only a proxy of energy consumption, internal parameters can be tuned to match the features of specific hardware, if known, achieving also absolute estimation.

Benchmark SNN models, both latency and rate based, exhibited a minimal loss in accuracy, compared with their equivalent ANNs, with significantly lower (from $-50\,\%$ to $-80\,\%$) EMAC per inference. An even larger improvement in energy consumption can be expected with SNNs when implemented on actual neuromorphic hardware, with respect to standard ANNs running on TPUs. While Surrogate Gradient proved to be an easy and effective way to achieve offline, supervised training of SNNs, scaling to very deep architectures, to recover state-of-the-art performances, is still an issue. This is in part due to the large amount of memory required at training: this value scales linearly with the size of the network (with constant number of time steps) and linearly with the number of time steps; but even if the final latency can be limited at the end of the training (especially with TTFS coding in which the inference ends as the first output spike is emitted), more deep networks require a higher latency at training, to allow a proper propagation of the spikes in the network: so, in practice, the memory used at training scales similarly to the square of the network size. This could limit the effectiveness of some regularization techniques, like BNTT: enabling training with limited memory can be achieved by reducing the batch size, which on the other hand can degrade the precision of statistical quantities like mean and variance of the input. Also, a higher number of time steps implies an accumulation over time of intrinsic gradient estimation errors due to surrogate gradient formulation, further lowering the training performance.

A research effort is still needed, especially in the search of new architectures capable to exploit SNNs peculiarities, and in the development of regularization techniques and initialization methods suited to latency-based networks. Attention should be also given to recent developments in online training techniques which do not require backward propagation in time, but only along the network at each time step [63], and new ANN-to-SNN conversion techniques tailored to achieve extremely low latency [64]. Overall, Spiking Neural Networks are a competitive candidate to achieve autonomy in space systems. For a successful application, future works should also explore sensitivity of neuromorphic processors and other event-based hardware to space environment, to identify possible disturbance models to be included in the training, enabling robustness even in presence of input or synaptic noise.

A. Test cases table

The complete list of the test cases run during the Aridna activity, together with their settings and parameters, is summarized in the following table. The notation adopted to describe the table field is detailed in Appendix B.

Table 3: Test cases and results.

Case	Type	Arch	Input size	Encoder	Structure	Neuron	Decoder	Decoding	dt (ms)	T	Regularization	Nl	Np	Test Acc.	Emitted spikes mean	Emitted spikes (std)	Latency mean	Latency (std)	E_{tot} (EMAC) mean	E_{tot} (EMAC) (std)
TEST_0	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(3), FC(128), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	64	—	3	6 013 092	0.671 11	390.0	(221.7)	64.0	(0.0)	2.8191e+06	(2.1354e+04)
TEST_1	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(5), FC(128), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	64	—	3	2 100 792	0.656 11	371.0	(223.3)	64.0	(0.0)	2.7490e+06	(2.0605e+04)
TEST_2	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(3), FC(128), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	64	—	3	6 013 092	0.665 19	366.8	(199.0)	64.0	(0.0)	2.8175e+06	(2.0732e+04)
TEST_3	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(5), FC(128), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	64	—	3	2 100 792	0.666 85	395.1	(216.2)	64.0	(0.0)	2.7489e+06	(2.0390e+04)
TEST_4	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(3), FC(32), FC(64), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	64	—	4	5 967 908	0.618 15	725.2	(304.4)	64.0	(0.0)	2.8122e+06	(1.9858e+04)
TEST_5	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(5), FC(32), FC(64), FC(10)	LIF	last_time_voltage_log	Rate	1	64	—	4	2 085 033	0.552 04	870.1	(573.3)	64.0	(0.0)	2.7481e+06	(2.0925e+04)
TEST_6	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(5), FC(32), FC(64), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	64	—	4	2 085 848	0.603 33	741.7	(313.5)	64.0	(0.0)	2.7433e+06	(1.9164e+04)
TEST_7	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(3), FC(64), FC(10)	LIF_trainable	last_time_voltage_log	Rate	10	64	—	3	5 981 220	0.529 26	2908.7	(611.1)	64.0	(0.0)	3.2530e+06	(1.1720e+05)
TEST_8	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(3), FC(128), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	128	—	3	6 013 092	0.711 67	543.7	(296.8)	128.0	(0.0)	5.6319e+06	(4.0391e+04)
TEST_9	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(3), FC(128), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	128	—	3	6 013 092	0.683 15	2082.4	(1 429.4)	128.0	(0.0)	5.6844e+06	(6.6741e+04)
TEST_10	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(3), FC(128), FC(128), FC(256), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	256	—	5	6 065 060	0.685 56	12 294.5	(5 486.5)	256.0	(0.0)	1.2482e+07	(3.5838e+05)
TEST_11	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(5), FC(128), FC(128), FC(256), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	256	—	5	2 152 760	0.665 93	12 164.6	(5 449.6)	256.0	(0.0)	1.2170e+07	(3.4204e+05)
TEST_12	SNN	MLP—LRF	3,64,64	ConstCurrentLIF (30, 0, 0.1)	LRF(5), FC(128), FC(128), FC(256), FC(10)	LIF_trainable	last_time_voltage_log	Rate	1	256	—	5	2 152 760	0.672 04	5171.2	(2 864.0)	256.0	(0.0)	1.1643e+07	(1.7653e+05)
P001	SNN	CONV	3,64,64	TTFSLinear (0.032)	C(16,3,1,1,zeros), MP(2), C(32,3,1,1,zeros), MP(2), C(64,3,1,1,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	Sakemi2021 (0.055, 70.0)	5	1 662 984	0.724 07	45 301.5	(1 562.3)	43.1	(2.8)	1.1407e+07	(3.0088e+05)
P002	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	MP(2), C(32,3,1,1,zeros), MP(2), C(64,3,1,1,zeros), FC(10) *	IFLOnce	ttfs_log	Latency	1	64	Sakemi2021 (0.055, 70.0)	4	187 328	0.809 44	11 898.6	(1 725.6)	41.2	(3.9)	9.6125e+06	(4.1941e+05)
P003	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	MP(2), C(32,3,1,1,zeros), MP(2), C(64,3,1,1,zeros), FC(10)	IFLOnce	ttfs_log	Latency	1	64	Sakemi2021 (0.055, 70.0)	4	187 424	0.828 33	10 976.1	(2 292.6)	38.2	(4.3)	8.7620e+06	(4.8070e+05)

(continued on next page)

Table 3, continued

Case	Type	Arch	Input size	Encoder	Structure	Neuron	Decoder	Decoding	dt (ms)	T	Regularization	Nl	Np	Test Acc.	Emitted spikes mean	Emitted spikes (std)	Latency mean	Latency (std)	E_{tot} (EMAC) mean	E_{tot} (EMAC) (std)
P004	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	MP(2), C(32,3,1,1,zeros), MP(2), C(64,3,1,1,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	Sakemi2021 (0.055, 70.0)	5	1 662 984	0.799 07	22 372.0	(1 332.2)	38.1	(2.5)	1.1343e+07	(2.7786e+05)
P005	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	Sakemi2021 (0.055, 70.0)	5	1 703 944	0.852 22	20 692.8	(1 458.5)	37.1	(2.4)	1.6175e+07	(5.1048e+05)
P006	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (neuron-wise), Sakemi2021 (0.055, 70.0)	5	4 849 672	0.849 44	15 272.5	(2 088.8)	41.0	(3.1)	1.4490e+07	(1.0759e+06)
P007	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	LIF	ttfs_log	Latency	1	64	Sakemi2021 (0.055, 70.0)	5	1 703 944	0.338 70	152 444.6	(28 323.7)	8.1	(1.3)	3.9955e+07	(4.7815e+06)
P008	SNN	CONV	3,64,64	ConvConstCurrentLIF (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	LIF	ttfs_log	Latency	1	64	Sakemi2021 (0.055, 70.0)	5	1 703 944	0.380 93	301 364.1	(144 284.0)	17.4	(6.9)	9.7982e+07	(2.9040e+07)
P009	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	LIF	max_voltage_log	Rate	1	64	—	5	1 703 944	0.835 93	377 620.6	(20 541.6)	64.0	(0.0)	1.0641e+08	(3.8253e+06)
P010	SNN	CONV	3,64,64	ConvConstCurrentLIF (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	LIF	max_voltage_log	Rate	1	64	—	5	1 703 944	0.812 41	864 987.6	(53 506.4)	64.0	(0.0)	3.1440e+08	(1.7705e+07)
P011	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(16,3,1,1), C(16,3,1,1), C(32,5,2,2), C(32,3,1,1), C(64,5,2,2), C(64,3,1,1), C(128,5,2,2), C(128,3,1,1), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	Sakemi2021 (0.055, 70.0)	11	1 288 072	0.111 85	87 184.9	(2 184.8)	64.0	(0.0)	4.6041e+07	(3.4887e+05)
P012	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(16,3,1,1), C(16,3,1,1), C(32,5,2,2), C(32,3,1,1), C(64,5,2,2), C(64,3,1,1), C(128,5,2,2), C(128,3,1,1), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (neuron-wise), Sakemi2021 (0.055, 70.0)	11	17 016 712	0.142 59	32 913.8	(16 018.7)	34.6	(13.2)	2.7176e+07	(3.4052e+06)
P013	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	Sakemi2021 (0.055, 70.0), Stanojevic2022 (0.001)	5	1 703 944	0.813 89	24 687.1	(1 177.1)	39.7	(2.8)	1.7593e+07	(4.6403e+05)
P014	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (neuron-wise), Sakemi2021 (0.055, 70.0), Stanojevic2022 (0.001)	5	4 849 672	0.815 37	13 882.4	(2 660.1)	41.8	(3.8)	1.2472e+07	(7.5213e+05)

(continued on next page)

Table 3, continued

Case	Type	Arch	Input size	Encoder	Structure	Neuron	Decoder	Decoding	dt (ms)	T	Regularization	Nl	Np	Test Acc.	Emitted spikes mean	Emitted spikes (std)	Latency mean	Latency (std)	E_{tot} (EMAC) mean	E_{tot} (EMAC) (std)
P016	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (neuron-wise)	5	4 849 672	0.850 37	15 988.5	(2 743.1)	42.0	(3.5)	1.3362e+07	(8.4222e+05)
P017	SNN	CONV	3,64,64	ConvConstCurrentLIF (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	LIF	max_voltage_log	Rate	1	64	—	5	1 703 944	0.813 15	804 631.1	(40 287.7)	64.0	(0.0)	3.0185e+08	(1.3671e+07)
P018	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (neuron-wise)	5	4 849 672	0.842 22	15 658.3	(3 119.0)	42.5	(3.1)	1.4066e+07	(9.1655e+05)
P019	SNN	CONV	3,64,64	ConvConstCurrentLIF (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	LIF	max_voltage_log	Rate	1	64	—	5	1 703 944	0.879 63	292 867.3	(99 414.0)	64.0	(0.0)	1.0862e+08	(2.5971e+07)
P020	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	LIF	ttfs_log	Latency	1	64	Sakemi2021 (0.055, 70.0)	5	1 703 944	0.833 52	113 548.8	(33 674.8)	35.0	(8.1)	3.5702e+07	(4.5138e+06)
P021	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	LIF	max_voltage_log	Rate	1	64	—	5	1 703 944	0.896 67	153 847.0	(8 911.4)	64.0	(0.0)	6.3803e+07	(2.5125e+06)
P022	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (neuron-wise)	5	4 856 822	0.756 30	14 920.3	(4 105.2)	41.6	(7.4)	1.5054e+07	(1.2918e+06)
P024	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (neuron-wise), Sakemi2021 (0.055, 70.0)	5	4 849 782	0.835 19	16 073.0	(3 239.6)	38.9	(2.6)	1.2853e+07	(1.1155e+06)
P025	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (neuron-wise), Sakemi2021 (0.055, 70.0)	5	4 856 182	0.847 22	17 511.1	(5 016.9)	39.8	(2.8)	1.3430e+07	(1.7634e+06)
P028	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	5	1 710 088	0.859 26	14 469.8	(1 228.1)	38.7	(2.4)	1.4149e+07	(5.1347e+05)
P029	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	5	1 710 088	0.886 48	14 841.0	(2 164.0)	37.5	(2.3)	1.3810e+07	(6.2981e+05)
P030	SNN	MLP	3,64,64	TTFSLinear (0.032)	FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	2	1 229 910	0.697 41	58.0	(5.4)	41.5	(3.5)	1.5061e+06	(5.7275e+04)
P031	SNN	MLP	3,64,64	TTFSLinear (0.032)	FC(200), FC(200), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	3	2 500 010	0.711 48	264.5	(15.7)	42.0	(4.5)	2.3682e+06	(7.3751e+04)
P032	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	5	1 710 088	0.906 11	19 244.6	(2 659.8)	36.4	(2.3)	1.2319e+07	(9.9517e+05)
P033	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(32,5,1,2,zeros), C(64,5,2,2,zeros), C(64,5,1,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	7	1 287 638	0.859 44	38 494.6	(4 497.0)	40.3	(2.1)	2.9616e+07	(2.7428e+06)
P034	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(64,5,2,2,zeros), C(128,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	5	3 521 128	0.852 96	35 373.2	(4 766.6)	39.0	(2.2)	3.4918e+07	(2.6614e+06)

(continued on next page)

Table 3, continued

Case	Type	Arch	Input size	Encoder	Structure	Neuron	Decoder	Decoding	dt (ms)	T	Regularization	Nl	Np	Test Acc.	Emitted spikes		Latency		E_{tot} (EMAC)	
															mean	(std)	mean	(std)	mean	(std)
P035	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(64,5,2,2,zeros), C(128,5,2,2,zeros), C(256,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	6	2 718 568	0.886 85	44 001.7	(2 688.3)	38.6	(2.1)	4.6846e+07	(1.7757e+06)
P036	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(64,5,2,2,zeros), C(128,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	5	3 521 238	0.909 81	39 504.2	(4 786.2)	35.9	(2.3)	2.9090e+07	(3.6289e+06)
P037	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(64,5,2,2,zeros), C(128,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	5	3 521 238	0.893 52	38 325.6	(3 021.7)	35.6	(2.3)	2.6249e+07	(2.8770e+06)
P038	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	5	1 710 198	0.879 44	18 643.6	(2 268.5)	36.5	(2.5)	1.4988e+07	(7.9992e+05)
P039	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(64,5,2,2,zeros), C(128,5,2,2,zeros), C(256,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	6	2 718 678	0.902 41	47 948.5	(7 764.5)	37.5	(2.3)	4.3057e+07	(5.0240e+06)
P040	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(64,5,2,2,zeros), C(128,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	5	3 521 238	0.902 96	38 159.6	(3 512.2)	35.7	(2.2)	2.9737e+07	(2.2662e+06)
P041	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	LIF	max_voltage_log	Rate	1	39	—	5	1 703 944	0.895 37	71 688.1	(8 635.4)	39.0	(0.0)	3.6394e+07	(2.4550e+06)
P042	SNN	CONV	3,64,64	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	LIF	max_voltage_log	Rate	1	64	BNTT (spatial),	5	1 710 088	0.731 48	504 926.0	(95 288.1)	64.0	(0.0)	1.0024e+08	(1.8734e+07)
P043	SNN	CONV	3,32,32	ConvIAFOnce (16, 3, 1, 1)	C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	5	481 288	0.837 59	6801.6	(409.5)	36.1	(3.6)	3.6992e+06	(1.2957e+05)
P044	SNN	CONV	3,32,32	ConvIAFOnce (20, 3, 1, 1)	C(40,5,2,2,zeros), C(20,5,2,2,zeros), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	4	57 260	0.800 93	5839.5	(414.8)	36.8	(4.8)	3.9688e+06	(3.1635e+05)
P045	SNN	CONV	3,32,32	ConvIAFOnce (32, 3, 1, 1)	C(64,5,2,2,zeros), C(128,5,2,2,zeros), FC(100), FC(10)	IFLOnce	ttfs_log	Latency	1	64	BNTT (spatial), Sakemi2021 (0.055, 70.0)	5	1 089 576	0.891 85	9568.0	(557.0)	37.9	(2.5)	1.1026e+07	(4.7378e+05)
PANN01	ANN	CONV	3,64,64	—	C(16,3,1,1,zeros), C(32,5,2,2,zeros), C(64,5,2,2,zeros), FC(100), FC(10)	ReLU	—	—	—	—	—	5	1 704 284	0.931 30	—	—	—	—	2.9623e+07	(0)
PANN02	ANN	MLP—LRF	3,64,64	—	LRF(3), FC(128), FC(10)	ReLU	—	—	—	—	—	3	63 406	0.660 00	—	—	—	—	7.6300e+04	(0)
PANN03	ANN	MLP—LRF	3,64,64	—	LRF(5), FC(128), FC(10)	ReLU	—	—	—	—	—	3	23 134	0.592 04	—	—	—	—	3.5587e+04	(0)
PANN04	ANN	MLP—LRF	3,64,64	—	LRF(3), FC(32), FC(64), FC(10)	ReLU	—	—	—	—	—	4	18 318	0.499 44	—	—	—	—	3.1244e+04	(0)
PANN05	ANN	MLP—LRF	3,64,64	—	LRF(5), FC(32), FC(64), FC(10)	ReLU	—	—	—	—	—	4	8286	0.395 93	—	—	—	—	2.0771e+04	(0)
PANN06	ANN	MLP—LRF	3,64,64	—	LRF(5), FC(64), FC(10)	ReLU	—	—	—	—	—	3	11 614	0.433 70	—	—	—	—	2.4131e+04	(0)

(continued on next page)

Table 3, continued

Case	Type	Arch	Input size	Encoder	Structure	Neuron	Decoder	Decoding	dt (ms)	T	Regularization	Nl	Np	Test Acc.	Emitted spikes mean	Emitted spikes (std)	Latency mean	Latency (std)	E_{tot} (EMAC) mean	E_{tot} (EMAC) (std)
PANN07	ANN	MLP—LRF	3,64,64	—	LRF(3), FC(128), FC(128), FC(256), FC(10)	ReLU	—	—	—	—	—	5	114 222	0.487 22	—	—	—	—	1.2673e+05	(0)
PANN08	ANN	MLP—LRF	3,64,64	—	LRF(5), FC(128), FC(128), FC(256), FC(10)	ReLU	—	—	—	—	—	5	73 950	0.543 33	—	—	—	—	8.6019e+04	(0)
PANN09	ANN	CONV	3,64,64	—	C(16,3,1,1), C(16,3,1,1), C(16,3,1,1), C(32,5,2,2), C(32,3,1,1), C(64,5,2,2), C(64,3,1,1), C(128,5,2,2), C(128,3,1,1), FC(100), FC(10)	ReLU	—	—	—	—	—	11	1 289 180	0.861 85	—	—	—	—	8.9097e+07	(0)
PANN10	ANN	MLP	3,64,64	—	FC(100), FC(10)	ReLU	—	—	—	—	—	2	1 229 916	0.649 44	—	—	—	—	1.2298e+06	(0)
PANN11	ANN	MLP	3,64,64	—	FC(200), FC(200), FC(10)	ReLU	—	—	—	—	—	3	2 500 016	0.692 96	—	—	—	—	2.4996e+06	(0)
PANN12	ANN	CONV	3,64,64	—	C(16,3,1,1,zeros), MP(2), C(32,3,1,1,zeros), MP(2), C(64,3,1,1,zeros), FC(100), FC(10)	ReLU	—	—	—	—	—	5	1 663 324	0.743 15	—	—	—	—	1.2846e+07	(0)
PANN13	ANN	CONV	3,64,64	—	C(16,3,1,1,zeros), C(64,5,2,2,zeros), C(128,5,2,2,zeros), FC(100), FC(10)	ReLU	—	—	—	—	—	5	3 509 372	0.943 89	—	—	—	—	8.3690e+07	(0)
PANN14	ANN	CONV	3,64,64	—	C(16,3,1,1,zeros), C(64,5,2,2,zeros), C(128,5,2,2,zeros), C(256,5,2,2,zeros), FC(100), FC(10)	ReLU	—	—	—	—	—	6	2 690 940	0.962 96	—	—	—	—	1.3448e+08	(0)
PANN15	ANN	CONV	3,32,32	—	C(20,3,1,1,zeros), MP(2), C(40,3,1,1,zeros), MP(2), C(20,3,1,1,zeros), FC(10)	ReLU	—	—	—	—	—	4	27 906	0.822 22	—	—	—	—	2.8698e+06	(0)
PANN16	ANN	CONV	3,32,32	—	C(32,3,1,1,zeros), C(64,5,2,2,zeros), C(128,5,2,2,zeros), FC(100), FC(10)	ReLU	—	—	—	—	—	5	1 077 852	0.942 04	—	—	—	—	2.7919e+07	(0)
dodo_a	SNN	CONV	3,32,32	—	C(20,3,1,1,zeros), MP(2), C(40,3,1,1,zeros), MP(2), C(20,3,1,1,zeros), FC(10)	LIF	max_voltage	Rate	1	32	—	4	18 140	0.917 22	40 850.6	(13 766.5)	32.0	(0.0)	5.7632e+06	(6.9816e+05)
dodo_b	SNN	MLP	3,16,16	—	FC(100), FC(200), FC(10)	IAFOnce	ttfs_neg	Latency	adim	—	Mostafa2017	3	98 800	0.682 59	264.0	(39.4)	755.1	(225.8)	4.5527e+05	(8.4250e+04)

(continued on next page)

Table 3, continued

Case	**Type**	**Arch**	**Input size**	**Encoder**	**Structure**	**Neuron**	**Decoder**	**Decoding**	**dt** (ms)	**T**	**Regularization**	**Nl**	**Np**	**Test Acc.**	**Emitted spikes** mean	(std)	**Latency** mean	(std)	**E_{tot}** (EMAC) mean	(std)
dodo_c	SNN	MLP	3,32,32	ConstCurrentLIF (100, 0, 0.2)	RecFC(100), FC(10)	LIF	max_voltage_log	Rate	1	32	—	2	656 400	0.732 04	3793.4	(616.1)	32.0	(0.0)	8.4346e+06	(1.2618e+06)

B. Test cases notation

In this section the notation adopted in Table 3 is explained. Each table field is expounded in the following.

Case

The identifier of the test case.

Type

```
SNN | ANN
```

Type of neural network (Spiking Neural Network or Artificial Neural Network).

Arch

Type of general architecture.

```
CONV
```

Convolutional Neural Network.

```
MLP
```

Multilayer Perceptron

```
MLP-LRF
```

MLP with limited receptive field (Sec. 7.3 at page 22).

Input size

```
nchannels,height,width
```

Size of the input image. If different from the original images in the EuroSat dataset, the file is properly scaled before to be fed to the network.

Encoder

Type of encoder:

```
ConstCurrentLIF (tau_syn, tau_mem, threshold)
```

Constant current LIF encoder (Sec. 4.1.1 at page 13). Some parameters to define the LIF neuron properties are needed in input.

```
TTFSLinear (tmax)
```

Linear TTFS encoder (Sec. 4.2.1 at page 4.2.1). `tmax` correspond to the maximum time to map the input as spike as in Eq. (4.1).

```
ConvIAFOnce (nchannels, kernelsize, stride, padding)
```

Learnable, convolutional variant (see Sec. 4.3 at page 15) of the Constant Current IAF encoder, with neurons spiking once at most (Sec. 4.2.2 at page 15). Convolution parameters are shown.

```
ConvConstCurrentLIF (nchannels, kernelsize, stride, padding)
```

Learnable, convolutional variant (see Sec. 4.3 at page 15) of the Constant Current LIF encoder (see above).

Structure

This field reports the actual structure of the network, describing all the sequence of the internal layers after the encoder (if any). In the following, the types of layer are summarized.

```
C (nchannels, kernelsize, stride, padding, padding_type)
```

Convolution layer.

```
FC (n_neurons)
```

Fully connected layer.

```
LRF (size_receptiveField)
```

Limited Receptive Field layer (see Sec. 7.3 at page 22).

```
MP (size)
```

Max Pool layer.

Neuron

Type of neuron adopted in teh network.

```
LIF
```

Leaky Integrate and Fire (Sec. 3.3 at page 11).

```
LIF_trainable
```

LIF neuron in which neuron internal parameters are actively learned and optimized during training.

```
IFLOnce
```

Non-leaky Integrate and Fire neuron with stepwise constant synapse (see Sec. 3.2 at page 3.2), which can spike once at most.

```
IAFOnce
```

Integrate And Fire neuron with direct synapse (see Sec. 3.1 at page 9), which can spike once at most.

```
ReLU
```

Standard ReLU activation function for ANNs.

Decoder

Specific type of decoder adopted at network output.

```
last_time_voltage_log
```

Last timestamp logarithmic voltage decoder (see Sec. 5.1 at page 17).

```
max_voltage_log
```

Maximum logarithmic voltage decoder (see Sec. 5.3 at page 17).

```
max_voltage
```

Maximum voltage decoder (see Sec. 5.2 at page 17).

```
ttfs_log
```

Logarithmic inverse Time-To-First-Spike decoder (see Sec. 5.5 at page 18).

```
ttfs_neg
```

Negative Time-To-First-Spike decoder (see Sec. 5.4 at page 17).

Decoding

```
Rate | Latency
```

Type of decoding: rate based or latency based, as detailed in Sec. 5 at page 17

dt `(float)`

Width of the adopted time step (ms). Time steps are often expressed with a physical unit of measure (ms), but for some architectures (i.e. case *dodo_b*) can be adimensional.

T `(int)`

Number of time steps in the simulation.

Reg loss

Regularization method applied to the model. More than one type of regularization can be applied to the same model.

```
BNTT (type)
```

Batch Normalization Through Time (see Sec. 6.3 at page 20). Two types of BNTT have been tested: neuron-wise or spatial.

```
Sakemi2021 (t_target, gamma)
```

Output spikes regularization for rank order coding like in [37] (see Sec. 6.1 at page 19). Target output time (`t_target`) and loss weights (`gamma`) are shown.

`Stanojevic2022 (gamma)`

Weight sum regularization as in [45] (see Sec. 6.2 at page 19). The value of the loss weight (`gamma`) is shown.

`Mostafa2017`

Weight sum regularization as in [36] (see Sec. 6.2 at page 19).

Nl `(int)`

Number spiking layers: it includes the encoder (if any).

Np `(int)`

Number of trainable parameters.

Test Acc. `(float)`

Classification accuracy evaluated on the test set.

Emitted spikes `(float, float)`

Emitted spikes per inference (mean value and standard deviation, evaluated on the test set). Inference is assumed to end at the Time-To-First-Spike at the output layer (for latency-based decoders); at the last time step T for rate-based decoding.

Latency `(float, float)`

Latency at inference (mean value and standard deviation, evaluated on the test set), expressed in number of time steps. Inference is assumed to end at the Time-To-First-Spike at the output layer (for latency-based decoders); at the last time step T for rate-based decoding. Time steps are often expressed with a physical unit of measure (ms), but for some architectures (i.e. case *dodo_b*) can be adimensional. Nevertheless, for the purposes of the energy consumption estimation, is the number of computed time step that matters, especially in a digital implementation in which the execution time does not necessarily coincide with the simulation time.

$E_{\textbf{tot}}$ `(float, float)`

Adimensional energy consumption per inference, expressed in Equivalent Multiply and ACcumulation floating point operation (EMAC) as detailed in Sec. 8 at page 23. The estimate is given as mean value and standard deviation. Inference is assumed to end at the Time-To-First-Spike at the output layer (for latency-based decoders); at the last time step T for rate-based decoding.

References

[1] Gianluca Giuffrida et al. "CloudScout: A Deep Neural Network for On-Board Cloud Detection on Hyperspectral Images". In: *Remote Sensing* 12.14 (2020). ISSN: 2072-4292. DOI: 10.3390/rs12142205. URL: https://www.mdpi.com/2072-4292/12/14/2205.

[2] Gianluca Furano et al. "Towards the Use of Artificial Intelligence on the Edge in Space Systems: Challenges and Opportunities". In: *IEEE Aerospace and Electronic Systems Magazine* 35.12 (Dec. 2020), pp. 44–56. ISSN: 1557-959X. DOI: 10.1109/MAES.2020.3008468.

[3] Michele Bechini, Michèle Lavagna, and Paolo Lunghi. "Dataset Generation and Validation for Spacecraft Pose Estimation via Monocular Images Processing". In: *Acta Astronautica* 204 (Mar. 1, 2023), pp. 358–369. ISSN: 0094-5765. DOI: 10.1016/j.actaastro.2023.01.012. URL: https://www.sciencedirect.com/science/article/pii/S0094576523000127 (visited on 01/13/2023).

[4] Stefano Silvestrini et al. "Implicit Extended Kalman Filter for Optical Terrain Relative Navigation Using Delayed Measurements". In: *Aerospace* 9.9 (2022). ISSN: 2226-4310. DOI: 10.3390/aerospace9090503.

[5] Stefano Silvestrini et al. "Optical Navigation for Lunar Landing Based on Convolutional Neural Network Crater Detector". In: *Aerospace Science and Technology* 123 (2022), p. 107503. ISSN: 1270-9638. DOI: 10.1016/j.ast.2022.107503.

[6] Guy Revach et al. "KalmanNet: Neural Network Aided Kalman Filtering for Partially Known Dynamics". In: *IEEE Transactions on Signal Processing* 70 (2022), pp. 1532–1547. ISSN: 1941-0476. DOI: 10.1109/TSP.2022.3158588.

[7] Arne Niitsoo et al. "A Deep Learning Approach to Position Estimation from Channel Impulse Responses". In: *Sensors* 19.5 (5 Jan. 2019), p. 1064. ISSN: 1424-8220. DOI: 10.3390/s19051064. URL: https://www.mdpi.com/1424-8220/19/5/1064 (visited on 06/28/2023).

[8] Bing Han et al. "Cross-Layer Design Exploration for Energy-Quality Tradeoffs in Spiking and Non-Spiking Deep Artificial Neural Networks". In: *IEEE Transactions on Multi-Scale Computing Systems* 4.4 (Oct. 2018), pp. 613–623. ISSN: 2332-7766. DOI: 10.1109/TMSCS.2017.2737625.

[9] Maxence Bouvier et al. "Spiking Neural Networks Hardware Implementations and Challenges: A Survey". In: *ACM Journal on Emerging Technologies in Computing Systems* 15.2 (Apr. 5, 2019), 22:1–22:35. ISSN: 1550-4832. DOI: 10.1145/3304103. URL: https://dl.acm.org/doi/10.1145/3304103 (visited on 06/29/2023).

[10] Saeed Reza Kheradpisheh and Timothée Masquelier. "Temporal Backpropagation for Spiking Neural Networks with One Spike per Neuron". In: *International Journal of Neural Systems* 30.06 (2020), p. 2050027. DOI: 10.1142/S0129065720500276.

[11] Aaron R. Voelker, Daniel Rasmussen, and Chris Eliasmith. *A Spike in Performance: Training Hybrid-Spiking Neural Networks with Quantized Activation Functions*. Mar. 1, 2021. DOI: 10.48550/arXiv.2002.03553. arXiv: 2002.03553 [cs, q-bio, stat]. URL: http://arxiv.org/abs/2002.03553 (visited on 06/29/2023). preprint.

[12] Christoph Stöckl and Wolfgang Maass. *Recognizing Images with at Most One Spike per Neuron*. 2020. DOI: 10.48550/arXiv.2001.01682. arXiv: 2001.01682 [cs]. URL: https://arxiv.org/abs/2001.01682. preprint.

[13] Saeed Reza Kheradpisheh, Maryam Mirsadeghi, and Timothée Masquelier. "BS4NN: Binarized Spiking Neural Networks with Temporal Coding and Learning". In: *Neural Processing Letters* 54.2 (Apr. 1, 2022), pp. 1255–1273. ISSN: 1573-773X. DOI: 10.1007/s11063-021-10680-x. URL: https://doi.org/10.1007/s11063-021-10680-x (visited on 06/29/2023).

[14] Eric Hunsberger and Chris Eliasmith. *Training Spiking Deep Networks for Neuromorphic Hardware*. 2016. DOI: 10.13140/RG.2.2.10967.06566. arXiv: 1611.05141 [cs]. URL: http://arxiv.org/abs/1611.05141 (visited on 01/20/2023). preprint.

[15] Seongsik Park et al. *T2FSNN: Deep Spiking Neural Networks with Time-to-first-spike Coding*. 2020. DOI: 10.48550/arXiv.2003.11741. arXiv: 2003.11741 [cs]. URL: http://arxiv.org/abs/2003.11741. preprint.

[16] Julian Göltz et al. *Fast and Energy-Efficient Neuromorphic Deep Learning with First-Spike Times*. May 17, 2021. DOI: 10.48550/arXiv.1912.11443. arXiv: 1912.11443 [cs, q-bio, stat]. URL: http://arxiv.org/abs/1912.11443 (visited on 12/22/2022). preprint.

[17] E.M. Izhikevich. "Which Model to Use for Cortical Spiking Neurons?" In: *IEEE Transactions on Neural Networks* 15.5 (Sept. 2004), pp. 1063–1070. ISSN: 1941-0093. DOI: 10.1109/TNN.2004.832719.

[18] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. "Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks". In: *IEEE Signal Processing Magazine* 36.6 (Nov. 2019), pp. 51–63. ISSN: 1558-0792. DOI: 10.1109/MSP.2019.2931595.

[19] Iulia M. Comsa et al. "Temporal Coding in Spiking Neural Networks with Alpha Synaptic Function". In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). May 2020, pp. 8529–8533. DOI: 10.1109/ICASSP40776.2020.9053856.

[20] Natalia Caporale and Yang Dan. "Spike Timing–Dependent Plasticity: A Hebbian Learning Rule". In: *Annual Review of Neuroscience* 31.1 (2008), pp. 25–46. DOI: 10.1146/annurev.neuro.31.060407.125639. pmid: 18275283. URL: https://doi.org/10.1146/annurev.neuro.31.060407.125639 (visited on 06/29/2023).

[21] Peter Diehl and Matthew Cook. "Unsupervised Learning of Digit Recognition Using Spike-Timing-Dependent Plasticity". In: *Frontiers in Computational Neuroscience* 9 (2015). ISSN: 1662-5188. DOI: 10.3389/fncom.2015.00099. URL: https://www.frontiersin.org/articles/10.3389/fncom.2015.00099.

[22] Amirhossein Tavanaei, Timothée Masquelier, and Anthony S. Maida. "Acquisition of Visual Features through Probabilistic Spike-Timing-Dependent Plasticity". In: *2016 International Joint Conference on Neural Networks (IJCNN)*. 2016 International Joint Conference on Neural Networks (IJCNN). July 2016, pp. 307–314. DOI: 10.1109/IJCNN.2016.7727213.

[23] Milad Mozafari et al. "Bio-Inspired Digit Recognition Using Reward-Modulated Spike-Timing-Dependent Plasticity in Deep Convolutional Networks". In: *Pattern Recognition* 94 (2019), pp. 87–95. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2019.05.015.

[24] Milad Mozafari et al. "SpykeTorch: Efficient Simulation of Convolutional Spiking Neural Networks With at Most One Spike per Neuron". In: *Frontiers in Neuroscience* 13 (2019), p. 625. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00625.

[25] Peter U. Diehl et al. "Fast-Classifying, High-Accuracy Spiking Deep Networks through Weight and Threshold Balancing". In: *2015 International Joint Conference on Neural Networks (IJCNN)*. 2015 International Joint Conference on Neural Networks (IJCNN). July 2015, pp. 1–8. DOI: 10.1109/IJCNN.2015.7280696.

[26] Wenzhe Guo et al. "Neural Coding in Spiking Neural Networks: A Comparative Study for Robust Neuromorphic Systems". In: *Frontiers in Neuroscience* 15 (2021), p. 212. ISSN: 1662-453X. DOI: 10.3389/fnins.2021.638474.

[27] Pritam Bose et al. “Spiking Neural Networks for Crop Yield Estimation Based on Spatiotemporal Analysis of Image Time Series”. In: *IEEE Transactions on Geoscience and Remote Sensing* 54.11 (Nov. 2016), pp. 6563–6573. ISSN: 1558-0644. DOI: 10.1109/TGRS.2016.2586602.

[28] Xiaoli Tao and Howard E. Michel. “Novel Artificial Neural Networks for Remote-Sensing Data Classification”. In: *Optics and Photonics in Global Homeland Security*. Optics and Photonics in Global Homeland Security. Vol. 5781. SPIE, May 19, 2005, pp. 127–138. DOI: 10.1117/12.609117. URL: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/5781/0000/Novel-artificial-neural-networks-for-remote-sensing-data-classification/10.1117/12.609117.full (visited on 06/29/2023).

[29] Edgar Lemaire et al. “An FPGA-Based Hybrid Neural Network Accelerator for Embedded Satellite Image Classification”. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020 IEEE International Symposium on Circuits and Systems (ISCAS). Oct. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180625.

[30] Shilpa Suresh, Devikalyan Das, and Shyam Lal. “A Framework for Quality Enhancement of Multispectral Remote Sensing Images”. In: *2017 Ninth International Conference on Advanced Computing (ICoAC)*. 2017 Ninth International Conference on Advanced Computing (ICoAC). Dec. 2017, pp. 9–14. DOI: 10.1109/ICoAC.2017.8441181.

[31] Friedemann Zenke and Surya Ganguli. “SuperSpike: Supervised Learning in Multilayer Spiking Neural Networks”. In: *Neural Computation* 30.6 (June 2018), pp. 1514–1541. ISSN: 0899-7667. DOI: 10.1162/neco_a_01086.

[32] Friedemann Zenke and Tim P. Vogels. “The Remarkable Robustness of Surrogate Gradient Learning for Instilling Complex Function in Spiking Neural Networks”. In: *Neural Computation* 33.4 (Mar. 2021), pp. 899–925. ISSN: 0899-7667. DOI: 10.1162/neco_a_01367.

[33] Julian Rossbroich, Julia Gygax, and Friedemann Zenke. “Fluctuation-Driven Initialization for Spiking Neural Network Training”. In: *Neuromorphic Computing and Engineering* (Oct. 2022). DOI: 10.1088/2634-4386/ac97bb;10.48550/arXiv.2206.10226.

[34] Sumit Bam Shrestha and Garrick Orchard. *SLAYER: Spike Layer Error Reassignment in Time*. Sept. 5, 2018. arXiv: 1810.08646 [cs, stat]. URL: http://arxiv.org/abs/1810.08646 (visited on 12/22/2022). preprint.

[35] Sander M. Bohte, Joost N. Kok, and Han La Poutré. “Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons”. In: *Neurocomputing* 48.1 (Oct. 1, 2002), pp. 17–37. ISSN: 0925-2312. DOI: 10.1016/S0925-2312(01)00658-0. URL: https://www.sciencedirect.com/science/article/pii/S0925231201006580 (visited on 06/30/2023).

[36] Hesham Mostafa. “Supervised Learning Based on Temporal Coding in Spiking Neural Networks”. In: *IEEE Transactions on Neural Networks and Learning Systems* 29.7 (July 2018), pp. 3227–3235. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2017.2726060.

[37] Yusuke Sakemi et al. “A Supervised Learning Algorithm for Multilayer Spiking Neural Networks Based on Temporal Coding Toward Energy-Efficient VLSI Processor Design”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2021), pp. 1–15. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2021.3095068.

[38] Patrick Helber et al. *EuroSAT: A Novel Dataset and Deep Learning Benchmark for Land Use and Land Cover Classification*. Feb. 1, 2019. arXiv: 1709.00029 [cs]. URL: http://arxiv.org/abs/1709.00029 (visited on 12/22/2022). preprint.

[39] Guillaume Bellec et al. *Long Short-Term Memory and Learning-to-Learn in Networks of Spiking Neurons*. Dec. 25, 2018. DOI: 10.48550/arXiv.1803.09574. arXiv: 1803.09574 [cs, q-bio]. URL: http://arxiv.org/abs/1803.09574 (visited on 12/22/2022). preprint.

[40] Steven K. Esser et al. "Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing". In: *Proceedings of the National Academy of Sciences* 113.41 (Sept. 2016), pp. 11441–11446. ISSN: 1091-6490. DOI: 10.1073/pnas.1604850113.

[41] Dongsung Huh and Terrence J Sejnowski. "Gradient Descent for Spiking Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc., 2018. URL: https://papers.nips.cc/paper_files/paper/2018/hash/185e65bc40581880c4f2c82958de8cfe-Abstract.html (visited on 06/30/2023).

[42] Stanisław Woźniak et al. "Deep Learning Incorporating Biologically Inspired Neural Dynamics and In-Memory Computing". In: *Nature Machine Intelligence* 2.6 (6 June 2020), pp. 325–336. ISSN: 2522-5839. DOI: 10.1038/s42256-020-0187-0. URL: https://www.nature.com/articles/s42256-020-0187-0 (visited on 06/30/2023).

[43] Daniel Auge et al. "A Survey of Encoding Techniques for Signal Processing in Spiking Neural Networks". In: *Neural Processing Letters* 53.6 (Dec. 1, 2021), pp. 4693–4710. ISSN: 1573-773X. DOI: 10.1007/s11063-021-10562-2. URL: https://doi.org/10.1007/s11063-021-10562-2 (visited on 06/28/2023).

[44] Sophie Denève and Christian K. Machens. "Efficient Codes and Balanced Networks". In: *Nature Neuroscience* 19.3 (3 Mar. 2016), pp. 375–382. ISSN: 1546-1726. DOI: 10.1038/nn.4243. URL: https://www.nature.com/articles/nn.4243 (visited on 06/28/2023).

[45] Ana Stanojevic et al. "Approximating Relu Networks by Single-Spike Computation". In: *2022 IEEE International Conference on Image Processing (ICIP)*. 2022 IEEE International Conference on Image Processing (ICIP). Oct. 2022, pp. 1901–1905. DOI: 10.1109/ICIP46576.2022.9897692.

[46] Youngeun Kim and Priyadarshini Panda. "Revisiting Batch Normalization for Training Low-Latency Deep Spiking Neural Networks From Scratch". In: *Frontiers in Neuroscience* 15 (2021). ISSN: 1662-453X. DOI: 10.3389/fnins.2021.773954. URL: https://www.frontiersin.org/articles/10.3389/fnins.2021.773954 (visited on 12/22/2022).

[47] Hesham Mostafa. *Supervised Learning Based on Temporal Coding in Spiking Neural Networks*. Aug. 16, 2017. DOI: 10.48550/arXiv.1606.08165. arXiv: 1606.08165 [cs]. URL: http://arxiv.org/abs/1606.08165 (visited on 12/22/2022). preprint.

[48] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. Version 3. Mar. 2, 2015. DOI: 10.48550/arXiv.1502.03167. arXiv: 1502.03167 [cs]. URL: http://arxiv.org/abs/1502.03167 (visited on 12/22/2022). preprint.

[49] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. "Training Deep Spiking Neural Networks Using Backpropagation". In: *Frontiers in Neuroscience* 10 (2016). ISSN: 1662-453X. URL: https://www.frontiersin.org/articles/10.3389/fnins.2016.00508 (visited on 06/22/2023).

[50] Christian Pehle et al. "The BrainScaleS-2 Accelerated Neuromorphic System With Hybrid Plasticity". In: *Frontiers in Neuroscience* 16 (2022). ISSN: 1662-453X. URL: https://www.frontiersin.org/articles/10.3389/fnins.2022.795876 (visited on 06/27/2023).

[51] Mike Davies et al. "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning". In: *IEEE Micro* 38.1 (Jan. 2018), pp. 82–99. ISSN: 1937-4143. DOI: 10.1109/MM.2018.112130359.

[52] Tien-Ju Yang et al. "A Method to Estimate the Energy Consumption of Deep Neural Networks". In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 2017 51st Asilomar Conference on Signals, Systems, and Computers. Oct. 2017, pp. 1916–1920. DOI: 10.1109/ACSSC.2017.8335698.

[53] Mark Horowitz. "1.1 Computing's Energy Problem (and What We Can Do about It)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). Feb. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[54] Brian Degnan, Bo Marr, and Jennifer Hasler. "Assessing Trends in Performance per Watt for Signal Processing Applications". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.1 (Jan. 2016), pp. 58–66. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2015.2392942.

[55] NVIDIA Corporation. *GP100 Pascal Whitepaper*. 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (visited on 06/30/2023).

[56] NVIDIA Corporation. *Nvidia Ampere GA102 GPU Architecture Whitepaper*. 2021. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf (visited on 03/03/2023).

[57] Mike Davies et al. "Advancing Neuromorphic Computing With Loihi: A Survey of Results and Outlook". In: *Proceedings of the IEEE* 109.5 (May 2021), pp. 911–934. ISSN: 1558-2256. DOI: 10.1109/JPROC.2021.3067593.

[58] Johannes Schemmel et al. *Accelerated Analog Neuromorphic Computing*. Mar. 26, 2020. arXiv: 2003.11996 [cond-mat, q-bio]. URL: http://arxiv.org/abs/2003.11996 (visited on 06/23/2023). preprint.

[59] Writam Banerjee, Revannath Dnyandeo Nikam, and Hyunsang Hwang. "Prospect and Challenges of Analog Switching for Neuromorphic Hardware". In: *Applied Physics Letters* 120.6 (Feb. 7, 2022), p. 060501. ISSN: 0003-6951. DOI: 10.1063/5.0073528. URL: https://doi.org/10.1063/5.0073528 (visited on 06/23/2023).

[60] Giacomo Indiveri et al. "Neuromorphic Silicon Neuron Circuits". In: *Frontiers in Neuroscience* 5 (2011). ISSN: 1662-453X. URL: https://www.frontiersin.org/articles/10.3389/fnins.2011.00073 (visited on 06/23/2023).

[61] Christian Pehle and Jens Egholm Pedersen. *Norse - A Deep Learning Library for Spiking Neural Networks*. Version 0.0.7. Oct. 2021. URL: https://github.com/norse/norse (visited on 06/27/2023).

[62] Paolo Lunghi and Stefano Silvestrini. *Investigation of Low Energy Spiking Neural Networks Based on Temporal Coding for Scene Classification – Study Proposal*. 2021.

[63] Bojian Yin, Federico Corradi, and Sander M. Bohté. "Accurate Online Training of Dynamical Spiking Neural Networks through Forward Propagation Through Time". In: *Nature Machine Intelligence* 5.5 (5 May 2023), pp. 518–527. ISSN: 2522-5839. DOI: 10.1038/s42256-023-00650-4. URL: https://www.nature.com/articles/s42256-023-00650-4 (visited on 08/22/2023).

[64] Zhanglu Yan et al. *HyperSNN: A New Efficient and Robust Deep Learning Model for Resource Constrained Control Applications*. Aug. 17, 2023. DOI: 10.48550/arXiv.2308.08222. arXiv: 2308.08222 [cs]. URL: http://arxiv.org/abs/2308.08222 (visited on 08/21/2023). preprint.

[65] L. Abbott, B. DePasquale, and RM Memmesheimer. "Building Functional Networks of Spiking Model Neurons". In: *Nature Neuroscience* 19 (2016), pp. 350–355. DOI: 10.1038/nn.4241.

[66] Guillaume Bellec et al. *Biologically Inspired Alternatives to Backpropagation through Time for Learning in Recurrent Neural Nets*. Feb. 21, 2019. DOI: 10.48550/arXiv.1901.09049. arXiv: 1901.09049 [cs]. URL: http://arxiv.org/abs/1901.09049 (visited on 12/22/2022). preprint.

[67] Guillaume Bellec et al. “A Solution to the Learning Dilemma for Recurrent Networks of Spiking Neurons”. In: *Nature Communications* 11.1 (1 July 17, 2020), p. 3625. ISSN: 2041-1723. DOI: 10.1038/s41467-020-17236-y. URL: https://www.nature.com/articles/s41467-020-17236-y (visited on 12/22/2022).

[68] Paolo Gabriel Cachi, Sebastián Ventura, and Krzysztof Jozef Cios. “Fast Convergence of Competitive Spiking Neural Networks with Sample-Based Weight Initialization”. In: *Information Processing and Management of Uncertainty in Knowledge-Based Systems*. Ed. by Marie-Jeanne Lesot et al. Cham: Springer International Publishing, 2020, pp. 773–786. ISBN: 978-3-030-50153-2. DOI: 10.1007/978-3-030-50153-2_57.

[69] Kristofor D. Carlson et al. “Biologically Plausible Models of Homeostasis and STDP: Stability and Learning in Spiking Neural Networks”. In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. 2013, pp. 1–8. DOI: 10.1109/IJCNN.2013.6706961.

[70] Iulia-Maria Comşa et al. “Spiking Autoencoders With Temporal Coding”. In: *Frontiers in Neuroscience* 15 (2021). ISSN: 1662-453X. URL: https://www.frontiersin.org/articles/10.3389/fnins.2021.712667 (visited on 12/22/2022).

[71] Gourav Datta, Souvik Kundu, and Peter A. Beerel. *Training Energy-Efficient Deep Spiking Neural Networks with Single-Spike Hybrid Input Encoding*. July 26, 2021. arXiv: 2107.12374 [cs]. URL: http://arxiv.org/abs/2107.12374 (visited on 07/04/2023). preprint.

[72] Mike Davies. “Taking Neuromorphic Computing to the Next Level with Loihi 2”. In: (2021).

[73] Jason K. Eshraghian et al. *Training Spiking Neural Networks Using Lessons from Deep Learning*. 2021. preprint.

[74] M. E. Fouda et al. *Spiking Neural Networks for Inference and Learning: A Memristor-based Design Perspective*. Oct. 8, 2019. DOI: 10.48550/arXiv.1909.01771. arXiv: 1909.01771 [cs]. URL: http://arxiv.org/abs/1909.01771 (visited on 02/13/2023). preprint.

[75] E. Paxon Frady et al. *Neuromorphic Nearest-Neighbor Search Using Intel’s Pohoiki Springs*. 2020. preprint.

[76] Brian Gardner and André Grüning. “Supervised Learning With First-to-Spike Decoding in Multilayer Spiking Neural Networks”. In: *Frontiers in Computational Neuroscience* 15 (2021). ISSN: 1662-5188. DOI: 10.3389/fncom.2021.617862.

[77] Dario Gil and William M. J. Green. “1.4 The Future of Computing: Bits + Neurons + Qubits”. In: *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*. 2020 IEEE International Solid- State Circuits Conference - (ISSCC). Feb. 2020, pp. 30–39. DOI: 10.1109/ISSCC19947.2020.9062918.

[78] Jesse Hagenaars, Federico Paredes-Vallés, and Guido de Croon. *Self-Supervised Learning of Event-Based Optical Flow with Spiking Neural Networks*. arXiv [Preprint] arXiv:2106.01862. 2021.

[79] Avi Hazan and Elishai Ezra Tsur. “Neuromorphic Analog Implementation of Neural Engineering Framework-Inspired Spiking Neuron for High-Dimensional Representation”. In: *Frontiers in Neuroscience* 15 (2021). ISSN: 1662-453X. URL: https://www.frontiersin.org/articles/10.3389/fnins.2021.627221 (visited on 07/04/2023).

[80] Geoffrey Hinton. *The Forward-Forward Algorithm: Some Preliminary Investigations*. Dec. 26, 2022. DOI: 10.48550/arXiv.2212.13345. arXiv: 2212.13345 [cs]. URL: http://arxiv.org/abs/2212.13345 (visited on 02/03/2023). preprint.

[81] Sebastian Höppner et al. “Dynamic Power Management for Neuromorphic Many-Core Systems”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.8 (Aug. 2019), pp. 2973–2986. ISSN: 1558-0806. DOI: 10.1109/TCSI.2019.2911898.

[82] Eric Hunsberger and Chris Eliasmith. *Spiking Deep Networks with LIF Neurons*. Oct. 29, 2015. DOI: 10.48550/arXiv.1510.08829. arXiv: 1510.08829 [cs]. URL: http://arxiv.org/abs/1510.08829 (visited on 06/30/2023). preprint.

[83] Dario Izzo et al. *Neuromorphic Computing and Sensing in Space*. Version 2. Dec. 17, 2022. DOI: 10.48550/arXiv.2212.05236. arXiv: 2212.05236 [cs]. URL: http://arxiv.org/abs/2212.05236 (visited on 01/17/2023). preprint.

[84] Geunyoung Kim et al. "Retention Secured Nonlinear and Self-Rectifying Analog Charge Trap Memristor for Energy-Efficient Neuromorphic Hardware". In: *Advanced Science* 10.3 (2023), p. 2205654. ISSN: 2198-3844. DOI: 10.1002/advs.202205654. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/advs.202205654 (visited on 06/23/2023).

[85] Youngeun Kim and Priyadarshini Panda. *Revisiting Batch Normalization for Training Low-latency Deep Spiking Neural Networks from Scratch*. Nov. 10, 2021. DOI: 10.48550/arXiv.2010.01729. arXiv: 2010.01729 [cs]. URL: http://arxiv.org/abs/2010.01729 (visited on 12/22/2022). preprint.

[86] Seijoon Kim et al. "Spiking-YOLO: Spiking Neural Network for Energy-Efficient Object Detection". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.07 (07 Apr. 3, 2020), pp. 11270–11277. ISSN: 2374-3468. DOI: 10.1609/aaai.v34i07.6787. URL: https://ojs.aaai.org/index.php/AAAI/article/view/6787 (visited on 02/10/2023).

[87] Andrzej S. Kucik and Gabriele Meoni. "Investigating Spiking Neural Networks for Energy-Efficient On-Board AI Applications. A Case Study in Land Cover and Land Use Classification". In: *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). Nashville, TN, USA: IEEE, June 2021, pp. 2020–2030. ISBN: 978-1-66544-899-4. DOI: 10.1109/CVPRW53098.2021.00230. URL: https://ieeexplore.ieee.org/document/9522999/ (visited on 01/13/2023).

[88] Chankyu Lee et al. "Enabling Spike-Based Backpropagation for Training Deep Neural Network Architectures". In: *Frontiers in Neuroscience* 14 (2020). ISSN: 1662-453X. URL: https://www.frontiersin.org/articles/10.3389/fnins.2020.00119 (visited on 07/13/2023).

[89] Erwann Martin et al. "EqSpike: Spike-driven Equilibrium Propagation for Neuromorphic Implementations". In: *iScience* 24.3 (Mar. 19, 2021), p. 102222. ISSN: 2589-0042. DOI: 10.1016/j.isci.2021.102222. URL: https://www.sciencedirect.com/science/article/pii/S2589004221001905 (visited on 06/29/2023).

[90] S. R. Nandakumar et al. "Experimental Demonstration of Supervised Learning in Spiking Neural Networks with Phase-Change Memory Synapses". In: *Scientific Reports* 10.1 (2020), p. 8080. DOI: 10.1038/s41598-020-64878-5.

[91] João D. Nunes et al. "Spiking Neural Networks: A Survey". In: *IEEE Access* 10 (2022), pp. 60738–60764. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3179968.

[92] Peter O'Connor et al. "Real-Time Classification and Sensor Fusion with a Spiking Deep Belief Network". In: *Frontiers in Neuroscience* 7 (2013), p. 178. ISSN: 1662-453X. DOI: 10.3389/fnins.2013.00178.

[93] Garrick Orchard et al. "Efficient Neuromorphic Signal Processing with Loihi 2". In: *2021 IEEE Workshop on Signal Processing Systems (SiPS)*. 2021 IEEE Workshop on Signal Processing Systems (SiPS). Oct. 2021, pp. 254–259. DOI: 10.1109/SiPS52927.2021.00053.

[94] Priyadarshini Panda et al. "ASP: Learning to Forget With Adaptive Synaptic Plasticity in Spiking Neural Networks". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8.1 (2018), pp. 51–64. DOI: 10.1109/JETCAS.2017.2769684.

[95] Roshani Pawar and Dr. S. S. Shriramwar. "Review on Multiply-Accumulate Unit". In: *International Journal of Engineering Research and Applications* 07.06 (June 2017), pp. 09–13. ISSN: 22489622, 22489622. DOI: 10.9790/9622-0706040913. URL: http://www.ijera.com/papers/Vol7_issue6/Part-4/B0706040913.pdf (visited on 07/04/2023).

[96] Gabriel Pereyra et al. *Regularizing Neural Networks by Penalizing Confident Output Distributions*. Jan. 23, 2017. DOI: 10.48550/arXiv.1701.06548. arXiv: 1701.06548 [cs]. URL: http://arxiv.org/abs/1701.06548 (visited on 02/03/2023). preprint.

[97] Nicolas Perez-Nieves et al. "Neural Heterogeneity Promotes Robust Learning". In: *Nature Communications* 5791.12 (2021). DOI: 10.1038/s41467-021-26022-3.

[98] Michael Pfeiffer and Thomas Pfeil. "Deep Learning With Spiking Neurons: Opportunities and Challenges". In: *Frontiers in Neuroscience* 12 (2018), p. 774. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00774.

[99] Seth Roffe et al. "Neutron-Induced, Single-Event Effects on Neuromorphic Event-Based Vision Sensor: A First Step and Tools to Space Applications". In: *IEEE Access* 9 (2021), pp. 85748–85763. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3085136.

[100] Bodo Rueckauer and Shih-Chii Liu. "Conversion of Analog to Spiking Neural Networks Using Sparse Temporal Coding". In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. May 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351295.

[101] Abhronil Sengupta et al. "Going Deeper in Spiking Neural Networks: VGG and Residual Architectures". In: *Frontiers in Neuroscience* 13 (2019), p. 95. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00095.

[102] Amirhossein Tavanaei et al. "Deep Learning in Spiking Neural Networks". In: *Neural Networks* 111 (Mar. 1, 2019), pp. 47–63. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2018.12.002. URL: https://www.sciencedirect.com/science/article/pii/S0893608018303332 (visited on 12/22/2022).

[103] Valerio Venceslai et al. "NeuroAttack: Undermining Spiking Neural Networks Security through Externally Triggered Bit-Flips". In: *2020 International Joint Conference on Neural Networks (IJCNN)*. July 2020, pp. 1–8. DOI: 10.1109/IJCNN48605.2020.9207351. arXiv: 2005.08041 [cs, stat]. URL: http://arxiv.org/abs/2005.08041 (visited on 12/22/2022).

[104] Xiaoyu Wang et al. "Advanced Optoelectronic Devices for Neuromorphic Analog Based on Low-Dimensional Semiconductors". In: *Advanced Functional Materials* 33.15 (2023), p. 2213894. ISSN: 1616-3028. DOI: 10.1002/adfm.202213894. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/adfm.202213894 (visited on 06/23/2023).

[105] Francis Wang et al. "Architecture-Level Energy Estimation for Heterogeneous Computing Systems". In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Mar. 2021, pp. 229–231. DOI: 10.1109/ISPASS51385.2021.00042.

[106] Kashu Yamazaki et al. "Spiking Neural Networks and Their Applications: A Review". In: *Brain Sciences* 12.7 (7 July 2022), p. 863. ISSN: 2076-3425. DOI: 10.3390/brainsci12070863. URL: https://www.mdpi.com/2076-3425/12/7/863 (visited on 02/17/2023).

[107] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. "Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). July 2017, pp. 6071–6079. DOI: 10.1109/CVPR.2017.643.

[108] Davide Zambrano and Sander M. Bohte. *Fast and Efficient Asynchronous Neural Computation with Adapting Spiking Neural Networks*. 2016. DOI: 10.48550/arXiv.1609.02053. arXiv: 1609.02053 [cs]. URL: https://arxiv.org/abs/1609.02053. preprint.

[109] Davide Zambrano et al. “Sparse Computation in Adaptive Spiking Neural Networks”. In: *Frontiers in Neuroscience* 12 (2019), p. 987. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00987.

[110] Friedemann Zenke. *Pub2018superspike*. Jan. 26, 2023. URL: https://github.com/fzenke/pub2018superspike (visited on 08/25/2023).

[111] Junhong Zhao et al. “Spiking Neural Network Regularization With Fixed and Adaptive Drop-Keep Probabilities”. In: *IEEE Transactions on Neural Networks and Learning Systems* 33.8 (Aug. 2022), pp. 4096–4109. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2021.3055825.