Spiking Neural Network-Based Near-Sensor Computing for Damage Detection in Structural Health Monitoring

MDPI

Article

Spiking Neural Network based Near-Sensor Computing for Damage Detection in Structural Health Monitoring

Francesco Barchi [1,†,*], Luca Zanatta [1,†], Emanuele Parisi [1,†], Alessio Burrello [1], Davide Brunelli [2], Andrea Bartolini[1] and Andrea Acquaviva [1]

[1] DEI, Università di Bologna
[2] DII, Università di Trento
* Correspondence: francesco.barchi@unibo.it
† These authors contributed equally to this work.

Abstract: In this work, we present an innovative approach for damage detection of infrastructures on edge devices, exploiting a brain-inspired algorithm. The proposed solution exploits recurrent Spiking Neural Network (LSNN), which are emerging for their theoretical energy efficiency and compactness, to recognise damage conditions by processing data from low-cost accelerometers (MEMS) directly on the sensor node. We focus on designing an efficient coding of MEMS data to optimise SNN execution on a low-power microcontroller. We characterised and profiled LSNN performance and energy consumption on a hardware prototype sensor node equipped with an STM32 embedded microcontroller and a digital MEMS accelerometer. We used a Hardware-in-the-Loop environment with virtual sensors generating data on an SPI interface connected to the physical microcontroller to evaluate the system with a data stream from a real viaduct. We exploited this environment also to study the impact of different on-sensor encoding techniques, mimicking a bio-inspired sensor able to generate events instead of accelerations. Obtained results show that the proposed optimised embedded LSNN (eLSNN), when using a spike-based input encoding technique, achieves 54% lower execution time with respect to a naive LSNN algorithm implementation present in the state-of-art. The optimised eLSNN requires around 47 kCycles, which is comparable with the data transfer cost from the SPI interface. However, the spike-based encoding technique requires considerably larger input vectors to get the same classification accuracy, resulting in a longer pre-processing and sensor access time. Overall the event-based encoding techniques leads to a longer execution time (1.49x) but similar energy consumption. Moving this coding on the sensor can remove this limitation leading to an overall more energy-efficient monitoring system.

Keywords: Spiking NN, SHM, Cyber-Physical Systems, Energy Efficiency, MEMS

Citation: Barchi, F.; Zanatta, L.; Parisi, E. et al. Spiking Neural Network based Near-Sensor Computing for Damage Detection in Structural Health Monitoring. *Future Internet* **2021**, *1*, 0. https://doi.org/

Received:
Accepted:
Published:

1. Introduction

The application of distributed sensors to pervasive monitoring of physical processes is one of the most critical and relevant domains. In particular, Structural Health Monitoring (SHM) is a key deployment scenario, where ensuring the safety of infrastructures such as buildings and bridges can be, in principle, achieved by deploying low-cost sensors to detect structural variations due to damages. The convergence of Artificial Intelligence (AI) to Internet-of-Things (IoT) and edge computing in this domain can help approach these challenges. In this context, executing AI detection algorithms directly on the IoT sensor nodes potentially reduces data transmission overheads and improves response time. A review of recent embedded AI approaches to detection in SHM can be found in [1]. Due to the low-cost, low-power and increasing accuracy of MEMS accelerometers, their application to distributed SHM is becoming popular.

Signal compression techniques on edge have also been proposed to compress MEMS data gathered from several nodes and sending them to the cloud storage and

analytic facility [2]. Still, on MEMS data, on-sensor modal estimation was proposed by implementing procedures to detect relevant peaks in the acquired signal spectrum [3].

Optimized machine learning algorithms on edge have been recently proposed to increase the intelligence of distributed detection for SHM. Considering the availability of code and libraries to implement machine learning algorithms on low-power microcontrollers [4,5], it is a viable solution to execute detection algorithms on-edge and on-sensor. In this context, hazard monitoring based on an array of event-triggered single-channel micro-seismic sensors with advanced signal processing is proposed exploiting a Convolutional Neural Network (CNN) implemented on a low-power microcontroller can be found in [6].

Applied to anomaly detection on a highway bridge, in [7] a compression technique to identify anomalies in the structure using a semi-supervised approach is proposed using either a fully connected or a convolutional autoencoder implemented on the sensor node. In the present work we provide an alternative solution applied to the same case study using a supervised algorithm for near-sensor anomaly detection based on Spiking Neural Networks (SNN). SNNs gained interest in the research community in various application domains, including SHM, because of their brain-inspired, event-based nature, which potentially allows reduced energy requirement compared to traditional ANN [8–12]. While Artificial Neural Networks (ANNs) have been successfully applied to SHM [1, 13–16], SNN are of increasing interest in this field because of their theoretical information greater processing efficiency by exploiting a sparse computation approach. In [17] a feed forward SNNs has been applied to low-cost, MEMS-based inspection of damaged buildings.

However, the state-of-art in SNN applications to SHM misses a real implementation of a data processing pipeline with the execution of SNN directly on the sensor node. Also, embedded machine learning libraries currently lack efficient SNN implementations. In the context of SNN, when time-series data from sensors are concerned, recurrent neural networks have shown to be effective [18]. For this reason, instead of more simple feed-forward architectures, we investigate recurrent SNNs for SHM data processing. In particular we consider a state-of-art recurrent implementation of SNN called LSNN (Long Short-Term SNN) introduced in [19] because of its interesting signal processing features and learning effectiveness. Also, a relevant aspect to be explored is the encoding of the input signal, which impacts subsequent computation steps and associated energy consumption. In particular, SNNs have been used with event-based input such as pixel variations from Dynamic Vision Sensor (DVS) cameras [20], but they can also effectively process "continuous" data streams in speech recognition applications [19]. However, in the context of SHM in general, which encoding is the best suited for anomaly detection task has not been studied so far.

This work presents the design, implementation, and characterisation of an LSNN on a low-power sensor node equipped with a commercial microcontroller and a MEMS accelerometer. The LSNN has been evaluated using real data from a highway viaduct, for which it was able to detect structural variations associated with a degraded condition. To the best of our knowledge, this is the first implementation of an LSNN on a low-power microcontroller that we integrated into a complete SNN-based near-sensor computing system. We designed and compared different input data encoding schemes in terms of performance and energy. We designed an optimised LSNN version for microcontroller targets, and we characterised its performance and energy consumption on silicon, including the overhead of data transfer from the MEMS sensor and their coding. Thanks to our Hardware-in-the-Loop measurement set-up, we were also able to emulate the behaviour of a smart sensor able to send spikes directly instead of acceleration values.

We are comparing with an alternative semi-supervised edge anomaly detection applied to the same dataset [7]. Authors of [7] show that the anomaly detection of the faulty and normal condition requires a complex pipeline. It consists of: i) A filtering step; ii) An anomaly detector; iii) A final smoothing post-processing. They either propose a

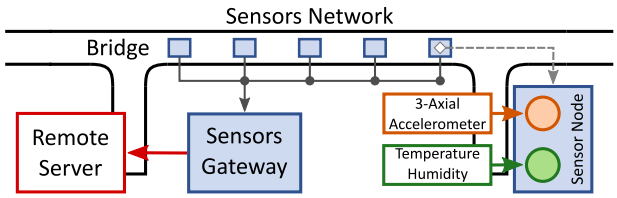

Figure 1. Monitoring system installation.

Principal Component Analysis compression and decompression or a fully connected autoencoder to implement the anomaly detector. The autoencoder features a single hidden layer of 32 neurons, an input layer with 500 samples, and an output layer with 500 neurons. Both these algorithms show a complexity of $500 \times 32 \times 2$ (32 000) multiply-and-accumulate operations[1]. We show that we can achieve similar accuracy results with considerably fewer computational resources, to solve the same detection problem, the proposed solution uses 15 750 sums and 750 multiplications.

While the two algorithms are not directly comparable because of the different ML approaches, the comparison against the reference testify that proposed solution is effective in solving the same detection problem. The contribution of this paper can be summarized as follows:

- We studied the computational requirements and complexity of the LSNN, and we provide an implementation on a low-power microcontroller-based sensor node.
- We designed and implemented an optimized LSNN version for performance constrained architectures, and we compared a continuous versus event-based input encoding.
- We evaluated the benefits of the proposed optimizations both theoretically and on real hardware, and we explored accuracy versus energy and performance trade-offs, including the cost of sensor data transfer.
- We demonstrate that LSNNs can be effectively executed near the MEMS sensor with a few tens of K cycles (comparable with data transfer cost) and deliver MCC levels higher than 0.75 (corresponding to almost 90% of accuracy) using data from a real case study of damage detection in SHM.

The rest of this paper is organized as follows. Section 2 presents some background on SHM and LSNN. Section 3 presents the LSNN architecture, training and input coding methods. Section 4 explains the introduced optimizations. Section 5 reports the experimental test-bed and results, and Section 6 concludes the work.

2. Background

This section describes the SHM problem and the reference SoA monitoring system composed of sensor nodes, edge-node, and cloud architecture. We then give some background about Spiking Neural Networks and their recurrent LSNN counterparts, input coding strategy, and training algorithm adopted. Finally, the sensor board and microcontroller used for measurements are introduced.

As described earlier, the manuscript focuses on the feasibility analysis of using a brain-inspired algorithm on the sensor-node MicroController Unit (MCU) and its implementation trade-offs. To experimentally validate this approach in Section 5 we will describe the experimental setup consisting of a hardware-in-the-loop (HIL) approach.

2.1. Bridge Structure & SHM Framework

The structure under study is a highway viaduct [2] built with eighteen sections, each one supported by two pairs of concrete pillars situated at their two ends. We

[1] The neurons, intended as the application in the hidden and output layers of a non-linear function like sigmoid or RELU, are 532 (hidden plus output neurons). Instead, the number of MAC operations is due to the number of connections. In this way, we have 500x32 MAC for the input-hidden layer and 32x500 MAC for the hidden-output layer, for a total of 500x32x2 MAC.

[2] A32 Torino-Bardonecchia - Viadotto S.S.335

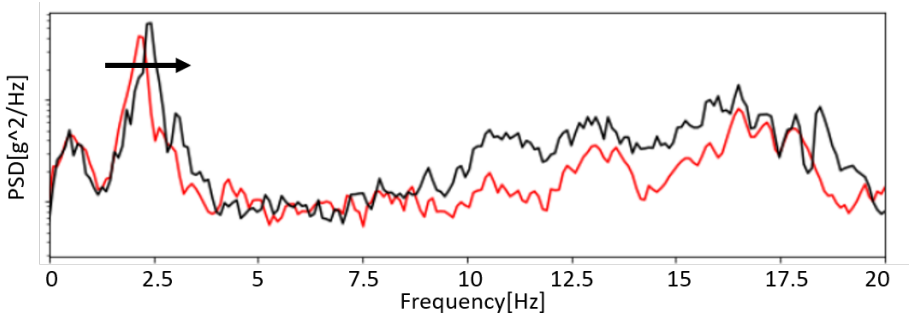

Figure 2. Mean natural frequency shift before and after scheduled maintenance.

focus on a single section instrumented for data analysis before a scheduled maintenance intervention in this work. Maintenance was necessary for the strengthening of the viaduct structure.

The acquisition framework is described in Figure 1. The system contains five identical sensor nodes. Each one features an STM32F4 microcontroller (MCU) and samples 3-axis accelerations and temperature data, and stores them into the cloud through a 4G-connected Raspberry Pi3 gateway.

The five nodes are connected through CAN-BUS to the gateway. Note that the acquisition system only collects the data without any local edge-side signal processing. In this basic configuration, data analysis is executed on the cloud. The accelerometer samples the data with a frequency of 25.6 kHz to avoid aliasing. Subsequently, the data is subsampled to have a final output frequency (f_s) of 100 Hz.

The cloud part is composed of a data-ingestion job, which receives and store data from the gateway, and periodically scheduled analysis tasks to monitor the health status of the bridge [21].

The MCU is an ARM 32-bit Cortex-M4 running at 168 MHz, with 192 kB of SRAM and 1 MB of Flash memory, popular in different edge applications for its low power consumption. Further, the MCU features a floating-point unit and a digital signal processing (DSP) library. The gateway is a standard Raspberry Pi 3 module B [22] (RPi3). It includes a Broadcom BCM2837 SoC, with 64 bits 4-core Cortex-A53 running at 1.2 GHz and 1 GB of DDR2 RAM. The gateway runs an Ubuntu operating system, easing the scheduling of communication tasks through common python interfaces (e.g., an MQTT broker [23]). The cloud system is divided into a storage section and a computing node allocated on the IBM cloud service.

In this work, we propose to replace cloud processing with a brain-inspired near-sensor anomaly detection algorithm directly computed on the microcontroller (MCU) on the sensor node board. We will show that the proposed algorithm can identify the normal and faulty bridge conditions, avoiding the RAW data transmission to the cloud.

2.2. Structural Health Monitoring Data

SHM frameworks usually process acceleration data for monitoring structures' health status [24]. The datasets used in the study contain 3-axial accelerations data acquired with the previously described framework. As above-mentioned, the viaduct underwent a technical intervention to strengthen its structure, with a corresponding change in the natural frequencies of a bridge. Figure 2 shows how the power spectrum density, averaged over 6 hours, is modified before and after the intervention.

Given these data's uniqueness, we use them as a proxy of an aged viaduct compared to a healthy one. In particular, in this work, we consider the signals collected after the intervention as the normal data produced by a healthy viaduct. Analogously, the data gathered before the maintenance intervention are considered "anomalies" since they are sampled on a damaged and aged bridge. Although the data do not represent the whole

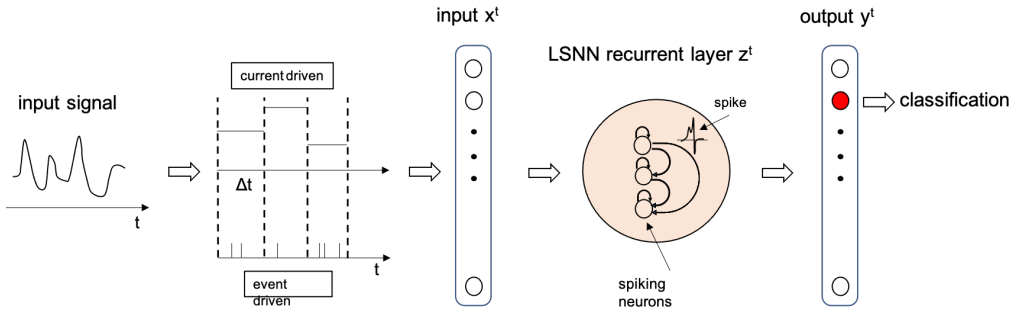

Figure 3. Full pipeline. The signal over time is encoded in current or events. Once transformed, it will be given as input to the SNN which will classify the data as a healthy or damaged bridge.

history of the viaduct, to the best of our knowledge, it is the only dataset containing vibrations from the viaduct during two different structural phases of the building.

2.3. Spiking Neural Networks

This paper proposes to study a Spiking Neural Network Model to solve the SHM supervised anomaly detection problem directly on the sensor's node MCU. The SNN model is a brain-inspired (so-called third-generation) type of neural network. They have a greater computational capacity as the single neuron is modelled with a much more complex dynamic than the neurons present in traditional Artificial Neural Networks (ANNs). This means that SNNs can solve the same tasks as ANNs with fewer neurons [8]. Moreover, their hardware implementation on neuromorphic architectures and accelerators can lead to greater energy efficiency in data management and computation [25–29]. In this work, we do not consider neuromorphic implementation because the objective is to work with low-cost commercial MCUs, for which SNN porting is not available.

In particular, we consider a recurrent type of SNN called LSNN (Long Short-Term SNN) because they are suitable to process temporal data streams like their artificial counterparts (e.g. LSTMs). While SNN has already attracted attention for SHM applications, so far, literature papers focused on simple feed-forward SNN, which are less powerful and do not exploit the potential of SNN, nor do they impose training challenges [17].

The recurrent LSNN structure is depicted in Figure 3, where the input, output and recurrent layers are represented. The input is a signal while the output is a classification encoded in the output neurons, meaning that each output neuron represents one of the possible classes. For instance, the neurons generating the highest output values is the one representing the recognised class. The time it takes to the network to process every single input and produce a stable output is called inference time (t_{inf}). As explained later in this Section, the input to the network can be either of current or event type. A current input type is a constant value over a time $\Delta t = t_{inf}$, while an event input type is a train of spikes, encoding the input signal using one of the possible methods described in Section 2.4.

This is the first work to study the feasibility of leveraging SNN inference on the sensor node on an embedded microcontroller. Detecting the health status of the structure directly on the node's MCU has the clear benefit of ease the network communication requirements of the sensors node to the edge node leading to energy reduction opportunities. The event-driven nature of the SNN processing can lead to an optimised implementation consisting of (i) a coding of the input sensor stream into a sequence of events depending on the intensity of inputs and (ii) a computation workload (internal activity of the SNN) which processes these events as spikes. Since spikes are binary signals, linear algebra operators can be implemented with simplified arithmetic. We designed a pre-processing stage to apply LSNN to SHM real-life dataset. The state-of-the-art of LSNN applied to a similar problem of phonemes recognition is solved in [19],

which processes the TIMIT dataset (representing phonemes) by Mel-frequency cepstrum (MFC). The spectral coefficients are given as input to the network as synaptic currents.

Starting from this reference LSNN (designed for server machines), we implemented an LSNN to process the spectral coefficient of the accelerometer waveform to detect a structural change in a highway viaduct. Through this network, we classified two categories of signals: Damaged or healthy (or repaired) bridge. In both cases, the bridge's natural frequency, detectable by oscillations due to the passage of vehicles, undergoes a shift that is typically difficult to identify in the presence of noise caused by environmental stimuli and variable traffic conditions. A spike neuron model is considerably more complex than an artificial neuron model (accumulation and threshold), so its training and inference require higher computational effort than its simplified version. To train the LSNN model, we applied Backpropagation Through Time (BPTT) algorithm [19].

The network used in this work is described in [19]. The input layer is composed of input neurons (I) which are connected in an all-to-all fashion through the W^I matrix to the recurrent layer, which is composed of Adaptative-Integrate and Fire neuron (ALIF). The recurrent layer is connected recursively to itself with an all-to-all connection matrix W^H, and it is linked to the output layer in an all-to-all fashion with the matrix W^O. The ALIF neurons are described by two state variables v and a. The first one is called membrane potential and increase when the neuron receives a stimulus (spike or current). When the v reach a value called v_{th}, it emits a spike. The ALIF neurons have a changeable v_{th}; this behaviour is described by a the second state variable. The following equations describe an ALIF neuron:

$$v_j^t = \overbrace{e^{-\frac{\delta t}{\tau_m}}}^{\alpha} v_j^{t-1} + \overbrace{\sum_{i \neq j} W_{ji}^H z_i^{t-1}}^{\text{ALif}\to\text{ALif}} + \overbrace{\sum_n W_{jn}^I x_i^t}^{\text{In}\to\text{ALif}} - \overbrace{v_{th} z_j^{t-1}}^{\text{Reset}} \tag{1}$$

$$a_j^t = \overbrace{e^{-\frac{\delta t}{\tau_a}}}^{\xi} a_j^{t-1} + z_j^{t-1} \tag{2}$$
$$A_j^t = v_{th} + \rho a_j^t$$

$$z_j^t = \begin{cases} 1 & \text{if } v_j^t \geq A_j^t \text{ and } r_j^t \neq 1 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Equation 1 describes the update of the membrane potential. α is the decay of the neuron, and it depends on the tick (δt) of the network and the membrane time constant τ_m. The second and the third term describe the contribution of the recurrent part and the input layer, respectively. In the end, there is the reset of the membrane potential if this reaches the v_{th}, z are the spikes of the ALIF, while x can be either spikes or current. Equation 2 describes the update of the spike threshold of the ALIF, ξ is the decay of the adaptative threshold, and it depends on δt and τ_a called decay time constant, a is rescaled by a factor ρ before being added to v_{th}. Equation 3 describe the spike condition. The neuron can spike (fire) only if it reaches a certain value (A) and if it is not in the refractory period (r). The refractory period is triggered when a neuron spike, and it is a time-lapse in which the neuron cannot fire.

The output neurons are continuous; therefore, the output is not a spike train but a continuous waveform. The following equation describes the outputs neurons:

$$y_k^t = \overbrace{e^{-\frac{\delta t}{\tau_o}}}^{\beta} y_k^{t-1} + \overbrace{\sum_j W_{kj}^O z_j^t}^{\text{ALif}\to\text{Out}} + \overbrace{b_k}^{\text{Bias}} \tag{4}$$

where τ_o is the decay constant of the membrane potential of the neurons and b is the bias of the neuron, which represents a constant current that stimulates the neuron.

2.4. Input Encoding Methods

The LSNN we used in this work can work with two types of input encodings: Current (current-driven) or events (events-driven). In the Current-Driven LSNN, the input signal is constant for all the inference time, while in the Events-Driven LSNN, the signal is first encoded as spikes and then provided as input. This section describes some of the state-of-the-art encoding methods and details the algorithm we implemented to encode acceleration signals coming from the MEMS sensor. Considering Figure 2, which shows the frequency shift that we want to detect, we give as input to the network the FFT of the acceleration signal due to the vehicle crossing the viaduct.

Literature is rich in algorithms to encode a waveform into a stream of spikes. Some of those approaches try to minimise signal reconstruction error, while others focus on emulating biological-plausible behaviours.

The authors of [30] propose a family of methods that minimise signal reconstruction error. All proposed methodologies are characterised by the presence of two complementary neurons (normally referred to as *positive* and *negative*), which expose a contrasting behaviour, that is, when one of the two fires, the other does not.

The simplest temporal encoding algorithm is called *Threshold Based Representation* (TBR). In this algorithm, whenever the difference between two consecutive signal samples is higher than a predetermined fixed threshold, then the *positive* neuron emits a spike. Unfortunately, while being computationally cheap to implement, TBR is known for leading to high reconstruction error [30], even for signals with simple dynamics.

The *Step Forward* (SF) method uses a baseline value (initialised as the value of the first signal sample) and a fixed threshold. Suppose the absolute value of the difference between two consecutive samples is higher than the sum of baseline and threshold. In that case, the *positive* neuron spikes and the baseline is updated, adding the baseline. A variation of this encoding strategy is called *Moving Window* (MW), where the baseline is updated looking at a moving window of signal samples. The other encoding methods proposed in [30] have not been considered in this work because of their inherent higher computational complexity that is not suitable for the chosen architectural target.

In [31], the authors describe some biological methods of encoding without considering the reconstruction error. In all these methods, the information is encoded in the reciprocal spikes of several neurons, meaning that proper encoding of the signal depends on the number of neurons adopted. At the time of the first spike, the information is stored in the delay between the start of the stimulus and the neuron's firing. In this method, the first neuron inhibits all the others; therefore, the information is in just one spike. In latency code, the information lies in the time between spikes of different neurons. In Rank-Order Coding (ROC), the information is encoded in the order of the spikes. In this method, every neuron can fire at most once for every sample (representing a single FFT in our case).

3. Brain Inspired Processing

This section describes the two main components of the brain-inspired processing pipeline, namely input encoding and LSNN architecture. Next section will describe the optimization performed to improve LSNN implementation for edge devices.

3.1. LSNN Input Coding

In the previous section, we discussed possible encoding methods proposed in the literature for LSNNs. In this section, we describe the method we applied to SHM data in our brain-inspired processing pipeline. An example of the input signal is shown in Figure 2.

In Figure 4 the entire data flow is shown. After collecting the data from the sensor, we extract only the z-axis values since they are more sensitive to the vehicles passages. These data are collected with a sampling frequency of 100 Hz and successively down-

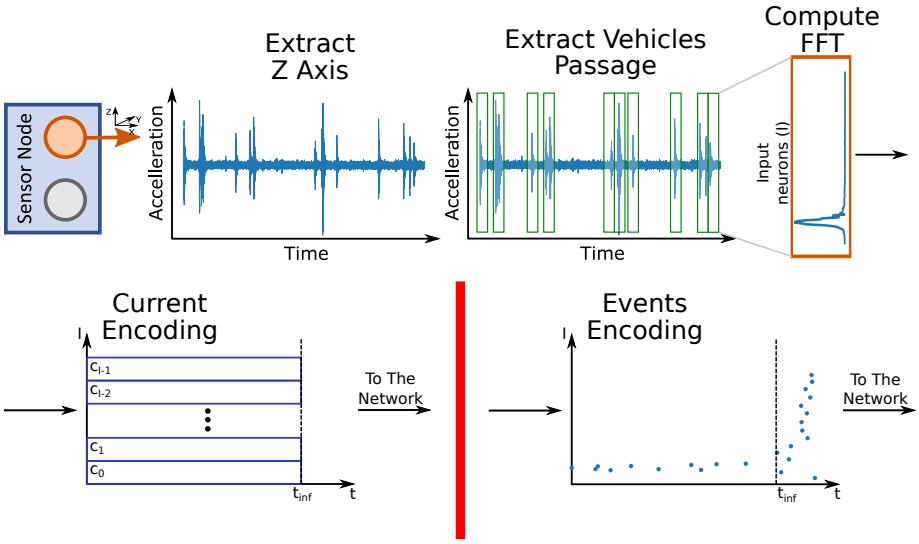

Figure 4. Data preprocessing.

sampled to 12.5 Hz. Subsequently, we get only the windows where vehicles passages have been detected. This was done by thresholding the signal.

The FFT of these windows is computed. The number of input neurons has been selected as two possible values: 50 or 150. This choice was made because a larger number of inputs would have caused a matrix of input weights (W^I) too large to train for such a small network, while a smaller number of inputs risks not extracting enough features from the signal [32]. So each vehicle's window can be composed of 100 or 300 coefficients that, in time, is equal to 8 or 24 seconds.

FFT coefficients are then given as input to the encoding stage. In the case of Current-Driven encoding, they are provided to the LSNN network one to each neuron for a constant time corresponding to t_{inf}. On the other side, in the case of Event-Driven encoding, the ROC algorithm is applied.

Each coefficient is thus encoded as a spike time interval. The larger the coefficient, the smaller the "time-to-spike". As a result, the spike time interval is inversely proportional to the value of the coefficients. As such, higher coefficients, which are more relevant for the damage detection because they have more energy, will fire first. Lower coefficients will fire later. Since t_{inf} is the inference time for each sample, all neurons that have not fired for $t < t_{inf}$ will no longer be able to fire. This implies that the coefficients with low energy (no information) will not impact the network input. The applied ROC encoding algorithm is shown below:

Algorithm 1: Compute Time-to-Event

input : A signal S of I coefficients and a inference time $tinf$
output: A Time-to-Event T array of I spiking intervals
$m = min(S)$;
$M = max(S)$;
for $i \leftarrow 0$ **to** $I - 1$ **do**
  $t = (S[i] - m)/(M - m)$;
  $tcheck = round(1/t)$;
  if $tcheck \leq tinf$ **then**
    T[i] = tcheck
  end
end
return T;

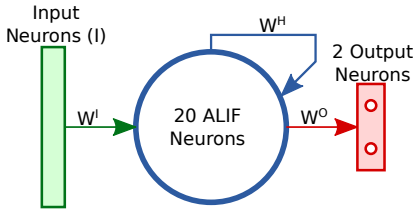

Figure 5. SNN architecture.

Parameter	Description	Explored Values
I	Input neurons	50, 150
τ_m	Recurrent layer membrane potential decay	20, 30
τ_o	Output layer membrane potential decay	3, 10, 30
v_{thc}	Spike threshold coefficient	0.01, 0.03
β_c	Adaptive threshold coefficient	1.7, 1.8
τ_{ac}	Adaptive threshold decay	0.5, 1.0
t_{inf}	Inference ticks	5, 10, 20
reg_c	Loss regularization coefficent	1, 100, 300
reg_r	Firing rate regularization coefficent	0.01, 0.001

Table 1: LSNN parameters explored

The signal S is the sequence of FFT coefficients corresponding to the observed acceleration window made of I coefficients. The inference time t_{inf} is also an input of the algorithm as it is a network parameter.

3.2. LSNN Architecture

The proposed LSNN network is shown in Figure 5, and it is composed of 20 ALIF and two output neurons which discriminate between the two classes of the bridge. We isolated the most important configuration parameters of the training algorithms (hyper-parameters) and studied their impact in terms of LSNN computational complexity and internal activity with an exhaustive search. The hyperparameters that we explored can be divided into two classes, the network parameters and the training parameters (summarised in Table 1). The first class comprises all the constants that strictly concern the network, while the second class is composed of all the values used to reach better performances but that don't feature the network. The network parameters that we explored are:

- The input number (I) [50, 150] is the number of input neurons in the first layer. A high number create a huge W^I matrix that cannot be trained by a small network, while a low number risks not extracting all the features of the input signal [32].
- τ_m [20, 30] represents the decay of the membrane potential in the recurrent layer. With high values, the neuron will need more time to return to the resting potential, representing the membrane potential of the neuron if any inputs have not perturbed it.
- τ_o [3, 10, 30] is the decay of the membrane potential of the output neurons.
- v_{thc} [0.01, 0.03]. The spike threshold (v_{th}) of the neurons is computing as: $v_{th} = v_{thc}/(1-e^{-\delta t/\tau_m})$. A high value of the v_{thc} is translated into a higher value of the threshold.
- β_c [1.7, 1.8] is used to compute the increase of the adaptive threshold (ξ) in the ALIF neurons. Higher values of this parameter are equivalent to a greater increase in the spike threshold.
- τ_{ac} [0.5, 1] is used to compute the decay of the adaptive threshold (τ_a). τ_a should have a value comparable to the time length of the problem [19].

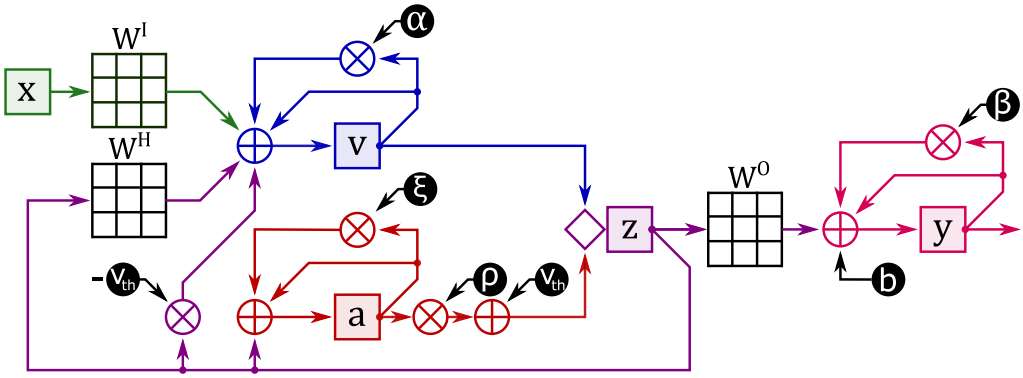

Figure 6. Flowchart of the LSNN update cycle. Neuron variables are represented with rectangles. Simple mathematical operations are represented with circles. Matrix multiplication operations with synaptic weights are represented by grids. Constants are represented with black circles. Data movements are represented by arrows between operators and operands.

The training parameters that we explored are the inference time (t_{inf}) expressed in ticks [5, 10, 20]. A tick in an SNN represents a cycle to complete the dynamics integration of all network neurons. The actual time depends on the implementation. The inference time is then the number of ticks that the network uses to process the input sample. For the regularisation coefficient we use (reg_c) [1, 100, 300] and for the regularisation rate we use (reg_r) [0.01, 0.001]. The two regularisation values are used to computing the loss of the network:

$$L_{fr} = \frac{1}{2}\sum_j \left(\frac{1}{nT}\sum_{t=1}^{nT} z_j^t - reg_r \right)^2 \tag{5}$$

$$L = L_p + L_{fr} * reg_c \tag{6}$$

Loss is composed of two parts (eq. 6), the loss of the classification (L_p) that is computing with the cross-entropy and the weighted firing-rate loss (L_{fr}). The L_{fr} is computed as the difference between the firing rate activity of the network and the reg_r that is the desired firing rate [19].

4. eLSNN: The Optimized Embedded LSNN

The implementation of LSNNs was performed taking into account resource constraints and exploiting SNN properties. In particular, sparsity in neuron response (e.g. firing) was exploited in order to skip processing cycles. The firing activity of neurons can be tuned as described in [19] using a loss value dedicated to limit the neuron activity.

Our LSNN implementation is depicted in Figure 6 and consists of four main steps: (i) Membrane voltage update ($\boldsymbol{v}$), (ii) Membrane voltage threshold update ($\boldsymbol{a}$), (iii) Evaluation of the spike emission ($\boldsymbol{z}$), (iv) Evaluation of the output ($\boldsymbol{y}$). The diagram shows: $\boldsymbol{x}$ data dependencies in green, $\boldsymbol{v}$ data dependencies in blue, $\boldsymbol{a}$ data dependencies in red, $\boldsymbol{z}$ data dependencies in purple and $\boldsymbol{y}$ data dependencies in magenta. In the next sections we will describe in detail the network implementation and the performed optimisations.

4.1. Membrane voltage update

The computation involved in the membrane voltage update ($\boldsymbol{v}$), as shown in Figure 6 in the first blue $\oplus$ operator, is done through the following steps:

1. Membrane voltage decay: $\boldsymbol{v^{(\alpha)}} := \alpha \boldsymbol{v}$
2. Contribution of inputs on membrane voltage: $\boldsymbol{v^{(I)}} := W^I \boldsymbol{x}$
3. Contribution of internal neuronal activity on membrane voltage: $\boldsymbol{v^{(H)}} := W^H \boldsymbol{z}$
4. Reduction of membrane voltage in case of previous spike emission: $\boldsymbol{v^{(th)}} := -v_{th} \boldsymbol{z}$

Item	Naive			Optimised		
	sum	mul	store	sum	mul	store
$v^{(\alpha)}$	-	1	-	-	1	-
$v^{(I)}$	I-1	I	-	$\tilde{x}$ - 1	-	-
$v^{(H)}$	N-1	N	-	$\tilde{z}$ - 1	-	-
$v^{(th)}$	-	1	-	1	-	-
v	4	-	1	4	-	1
Total	I+N+2	I+N+2	1	$\tilde{x}+\tilde{z}+3$	1	1

Table 2: Operations analysis for a single neuron. These operations must be executed for each neuron (N) and for each inference tick (t_{inf}).

5. Update of membrane voltage: $\boldsymbol{v} \leftarrow \boldsymbol{v} + \boldsymbol{v}^{(\alpha)} + \boldsymbol{v}^{(I)} + \boldsymbol{v}^{(H)} + \boldsymbol{v}^{(th)}$

In Table 2 we report the operations (sum and multiply) of both the non optimized (Naive) and optimised version for each component of $\boldsymbol{v}$. Overall, the operations to be performed are: $(I+N+2)Nt_{inf}$ sum and $(I+N+2)Nt_{inf}$ multiplication.

Because of the event-driven design of the network, it is possible to implement the computation of the $v^{(I)}$ and $v^{(H)}$ components more efficiently. Considering the row-column multiplication between the matrix W (with $N \times M$ elements) and the column vector $\boldsymbol{x}$ (with $M \times 1$ elements), the operation result will be added to the vector $\boldsymbol{v}$. When the elements of $\boldsymbol{x}$ can only assume binary values ($x_i \in \{0,1\}$) the $W\boldsymbol{x}$ operation can be implemented using only sums. The following pseudo-code formally describes the operation:

Algorithm 2: Optimized implementation of $v \leftarrow W\boldsymbol{x}$ when x is a boolean vector.

```
i ← 0;
while i < events do
    offset ← x̆[i];
    while n ≤ neurons do
        v[n] ← v[n] + W[offset];
        offset ← offset + neurons;
    end
    i ← i+1;
end
```

Let $\breve{\boldsymbol{x}}$ be a list containing only the index of the non-zero elements of $\boldsymbol{x}$. By iterating over the elements of $\breve{\boldsymbol{x}}$ we select the columns of W to be considered in the sum cycle (inner cycle). The sum cycle iterates over the rows of the selected column, and for each element accumulates its content in the result vector. The required operations will therefore be dependent on the number of non-zero elements in the vector $\boldsymbol{x}$. The number of non-zero elements in the vector $\boldsymbol{x}$ will change with each inference tick. We then identify using the symbol $\tilde{\boldsymbol{x}}$ the average number of non-zero elements of $\boldsymbol{x}$ for each inference tick.

We then reduce the sums required for the membrane voltage update from $(I+N+2)Nt_{inf}$ to $(\tilde{\boldsymbol{x}}+\tilde{\boldsymbol{z}}+\mathbf{3})Nt_{inf}$ and the multiplications from $(I+N+2)Nt_{inf}$ to Nt_{inf}. The value of $\tilde{\boldsymbol{x}}$ depends on the chosen event encoding. The value of $\tilde{\boldsymbol{z}}$ depends on the behaviour of the network. When training the network, it is possible to minimise $\tilde{\boldsymbol{z}}$ by introducing its value into the loss calculation.

4.2. Threshold update

Updating the threshold for the membrane voltage ($\boldsymbol{a}$), as shown in Figure 6 in the first red $\oplus$ operator, is broken down into the following operations:

1. Evaluation of threshold decay: $\boldsymbol{a}^{(\xi)} := \xi \boldsymbol{a}$
2. Threshold adjustment in case of previous spike: $\boldsymbol{a}^{(z)} := z$
3. Threshold update: $\boldsymbol{a} \leftarrow \boldsymbol{a} + a^{(\xi)} + a^{(z)}$
4. Weighted threshold computation: $a^{(\rho)} := \rho a$
5. Reference threshold augmentation $v_{th}^{(a)} := v_{th} + a^{(\rho)}$

In total, $3Nt_{inf}$ sums and $2Nt_{inf}$ multiplications are required to implement the adaptive threshold functionality.

4.3. Spike emission check

At each inference tick, the spike firing condition must be checked. For each neuron, the membrane voltage is compared with the spike threshold. In this particular implementation, the threshold voltage is adapted to the activity of the neuron and increases as its activity increases. A spiking neuron is also inhibited for a period t_{ref} called refractory time. Even if the neuron has a membrane voltage above the threshold during this period, it will not fire. To check this condition, each neuron stores the information about the tick of the last spike $t_i^{(z)}$.

When checking the emission of a spike, as shown in Figure 6 in the first purple $\Diamond$ operator, two conditions must therefore be checked for each neuron:

1. The membrane voltage must be above the adaptive threshold: $v_i \geq v_{th}^{(a)}$
2. The neuron must not be inhibited by an earlier spike: $t \geq t_i^{(z)} + t_{ref}$

If the above conditions are satisfied, the vector $\boldsymbol{z}$ will take the value 1 at position i, otherwise the value 0.

In our implementation, instead of directly handling the vector $\boldsymbol{z}$, we use the list of events $\check{\boldsymbol{z}}$. Using the list of events, we can replace matrix multiplications $W^H\boldsymbol{z}$ and $W^O\boldsymbol{z}$ with sums. At the beginning of the spike emission check phase, the list $\check{\boldsymbol{z}}$ of previous events are cleared. In the presence of a spike emission, the identifier of the spiking neuron will be added into the $\check{\boldsymbol{z}}$ list.

4.4. Output update

Output neurons receive spikes generated by network neurons in the recurrent layer at the same tick as they are emitted. As shown in Figure 6 in the first magenta $\oplus$ operator, the output neurons have a similar update cycle to the recurrent neurons:

1. Output decay: $\boldsymbol{y}^{(\beta)} := \beta \boldsymbol{y}$
2. Contribution of internal neuronal activity on output: $\boldsymbol{y}^{(\boldsymbol{H})} := W^O\boldsymbol{z}$
3. Bias contribution: $\boldsymbol{y}^{(\boldsymbol{b})} := \boldsymbol{b}$
4. Update of output: $\boldsymbol{y} \leftarrow \boldsymbol{y} + \boldsymbol{y}^{(\beta)} + \boldsymbol{y}^{(\boldsymbol{H})} + \boldsymbol{y}^{(\boldsymbol{b})}$

Again, the calculation of $\boldsymbol{y}^{(\boldsymbol{H})}$ requires a multiplication between the matrix W^O and the vector $\boldsymbol{z}$. The procedure described in 4.1 helps to lower the number of operations. Using the list $\check{\boldsymbol{z}}$ it is possible to solve the operation $W^O\boldsymbol{z}$ using sums only. The number of operations is then lowered from $(N-1)Ot_{inf}$ sums and NOt_{inf} multiplications to $\check{z}Ot_{inf}$ sums.

4.5. Current-driven input

In this work, we consider also an alternative to input events, using continuous (e.g. real) values in the vector $\boldsymbol{x}$ [19]. While in this case it is not possible to perform the optimisations described in Algorithm.2 for solving $\boldsymbol{v}^{(I)} := W^I\boldsymbol{x}$, we provide an optimised version of the LSNN using this type of input to evaluate the trade-offs lead by the two encoding methods and reported in Section 5. We note that in this case the contribution of internal neuronal activity $\boldsymbol{v}^{(H)}$ (hidden/recursive layer) is still spike-based, however in the input-layer it is not possible to remove multiply operations as it is possible using input spike events.

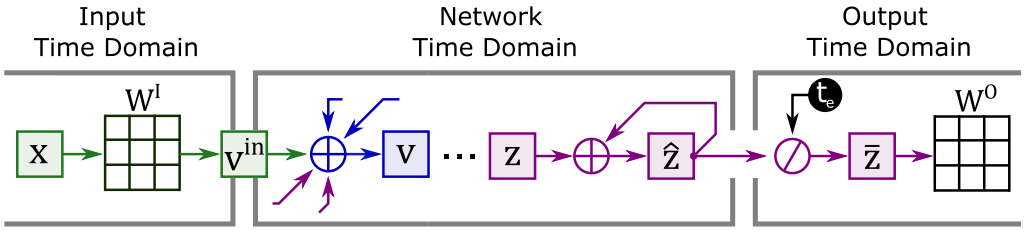

Figure 7. Flowchart modified to handle current-driven input. In this application scenario the input and output have two different time domains from the network. v^{in} and $\hat{z}$ vectors are used to pass information from the input to the network and from the network to the output.

In the literature, networks using continuous input typically observe a sample for a period called the exposure time t_{exp}. Within the network exposure time, the same values of $\boldsymbol{x}$ are always presented. This leads to the formation of three time domains (although it is common for the time domains of input and output to coincide) represented in figure 7:

- Temporal domain of the input, where the input varies over time.
- Temporal domain of the network, for each tick in the input-time-domain the network performs t_{exp} iterations always observing the same value of $\boldsymbol{x}$
- Temporal domain of output, where output varies over time.

In order to efficiently handle this LSNN variant, the time domains are decoupled by vectors $\boldsymbol{v^{in}}$ and $\hat{\boldsymbol{z}}$. The first vector decouples the input from the network; it is computed by a row-column multiplication between the matrix W^I and the vector $\boldsymbol{x}$ each time the input changes. Then, the vector $\boldsymbol{v^{in}}$ will be used to increment $\boldsymbol{v}$ at each tick of the network. The number of operations to compute $\boldsymbol{v}^{(I)}$ then becomes $({}^1/_{t_{exp}}(I-1)N+N)t_{inf}$ sums and ${}^{t_{inf}}/_{t_{exp}}IN$ multiplications.

The second vector decouples the network from the output. The output will no longer process the spikes coming from the network but will use the average activity of the network within the exposure time as information. At each tick, the network must accumulate the emitted spikes inside the vector $\hat{\boldsymbol{z}}$. In the time domain of the output the content of the vector will be averaged ($\bar{\boldsymbol{z}}$) and used for the calculation of $\boldsymbol{y^{(H)}} := W^O\bar{\boldsymbol{z}}$.

The number of operations for calculating $\boldsymbol{y^{(H)}}$ then becomes $({}^1/_{t_{exp}}(N-1)O+\tilde{\boldsymbol{z}})t_{inf}$ sums and ${}^{t_{inf}}/_{t_{exp}}NO$ multiplications.

In the following, we will refer to this LSNN version as *current-driven* eLSNN, since the continuous and constant input over the exposure time can be interpreted as a constant current stimulus. The version working with input spikes (e.g. binary values) will be referred to as *event-driven* eLSNN instead.

5. Experimental Result

This section first describes the experimental setup used to test the LSNN implementation presented in the previous sections. Then, it reports the results of the design trade-off characterisation for the proposed eLSNN. In particular:

- Section 5.1 describes the Hardware-in-the-Loop system we implemented to profile the runtime and energy performance of the SHM application.
- In Section 5.2, we study the accuracy of the trained eLSNNs (current-driven and event-driven). We evaluated the MCC on the test dataset for the first order hyper-parameters, which impact the energy and computational efficiency of the eLSNN implementation. The first-order hyper-parameters are the input number (I) and inference ticks (t_inf) of the networks.
- In Section 5.3, we study the impact of the eLSNN performance (*#execution cycles*) considering the most accurate networks for each eLSNN version (current-driven and event-driven). We also considered different combinations of the first-order parameters as a function of the activity factors of the eLSNNs (number of non-zero elements (spikes), both in the input (x) and hidden/recurrent layer z).

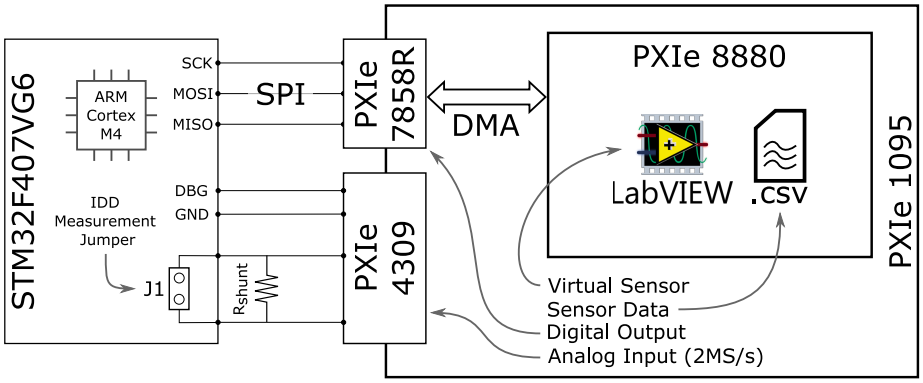

Figure 8. Architecture of the Hardware-in-the-Loop test system. The PXI system comprises a PXIe-4309 *Analog Input* module to monitor current consumption in each stage of the pipeline. The PXIe7858R FPGA, along with the PXIe-8880 Windows 7 controller, implements the *Virtual Sensor* and allow loading MEMS readings from a CSV file in real-time, emulating a real buffered digital MEMS node in hardware. The whole application runs on a STM32F4 development board featuring a Cortex-M4 ARM core.

- In Section 5.4, we perform a complete characterisation of the SHM sensing node firmware for a network implementation having median activity-factors. This corresponds to a typical behaviour of the sensor node for a real SHM application in terms of execution time and energy-consumption.

5.1. Testbed

We prepared a Hardware-in-the-Loop setup to characterise the performance and energy of the data acquisition and processing board. As shown in Figure 8, the testbed we implemented is composed of three actors: i) A development board hosting the STM32 MCU (*System-on-Chip*) on the left; ii) A data acquisition system to monitor the consumption of the STM32 development board (*Analog Input*) on the right and iii) A FPGA emulation module able to feed the processing unit in the system-on-chip with data coming from the real-world SHM application described in Section 2 in real-time (*Virtual Sensor*). (based on PXIe systems from NI [33])

Both the *Analog Input* and the *Virtual Sensor* are part of a PXIe systems from NI [33]). This is a modular, LabVIEW controlled environment able to perform fast measurements requiring high-performance digital and analog I/O. The following three subsections provide further details about the PXI modules used in the experimental setup and their role in the pipeline characterisation.

5.1.1. System-on-Chip

We implemented and deployed the entire SHM pipeline on an STM32F407VG6 MCU. It is a system-on-chip manufactured by STMicroelectronics, which features an ARM Cortex-M4 with a floating-point unit. The core comes with 192 KB RAM and 1 MB FLASH memory and supports a maximum clock frequency of 168 MHz. The STM32F407VG6 supports multiple low-power modes and various clock frequency configurations, which allow a fine-grained tuning of the system performance depending on the application needs. For the sake of this work, we apply the following policies:

- When the system is in RUN mode, the system-on-chip is always clocked at the maximum allowed clock frequency, equal to 168 MHz.
- When the system fetches data from the MEMS sensor over SPI, the transfer is performed via DMA with the core in SLEEP mode. Before entering SLEEP mode, the MCU core is clocked down to 16 MHz to minimize SLEEP current consumption.
- When the system-on-chip is not working, it is put in STANDBY mode, switching off the voltage regulator to achieve the lowest consumption possible. Notice that

we can apply such an aggressive power-saving policy since we assume our SHM application does not need to retain any state before one LSNN inference and the next inference.

Considering the software stack employed to implement the SHM application, we interfaced the on-board peripherals using the Hardware Abstraction Layer provided by STMicroelectronics. Instead, the mathematical processing (e.g. FFT computation, FFT-to-spike conversion) is implemented using the CMSIS-DSP library primitives. It is a set of routines developed by ARM to deliver DSP-like functions optimized to run on Cortex cores.

5.1.2. Analog Input

The power and performance monitoring activity of the pipeline has two requirements: i) Measure the current sunk by the system-on-chip at any time of the data acquisition and processing and ii) precisely split the current waveform into each pipeline stage, to detect the analysis stages that are the more power hungry or require most MCU cycles to be accomplished. Both requirements are met by using a PXIe-4309 ADC device reading 2M Samples per second and featuring 32 channels 18 bit wide. We used two of the available input channels to perform the following synchronous activities:

- Sample the current sunk by the MCU using a 1 Ohm shunt resistor. Such measurement exploits a jumper available on the development board to monitor current consumption on the VDD of the system-on-chip.
- Read the logic level of a debug GPIO (DBG pin, Figure 8) that is toggled by the application software each time a processing stage is started or completed, to associate each phase (e.g. SPI transfer, FFT computation, SNN inference) to its current consumption waveform (with negligible overhead on the application performance).

5.1.3. Virtual Sensor

The *Virtual Sensor* is an FPGA-based emulation system that allows feeding the SPI interface of the MCU with data taken from a trace. The trace was obtained from a real sensor deployment on the field. For characterisation and profiling purposes, the virtual sensor was connected to the MCU replacing the MEMS present in the sensor node board. The *Virtual Sensor* is completely implemented within the PXI system and is made of two modules sharing data through a DMA-controlled hardware FIFO:

- PXIe-8880: The PXI system controller, which is a general-purpose CPU-based host running Windows 7 and LabVIEW. It loads the accelerations measured on the bridge from a CSV file and pushes them into a FIFO at the boundary of the FPGA system.
- PXIe-7858R: The PXI FPGA module, which loads the measurements the DMA move from the controller to its FIFO. It acts as a digital MEMS that samples structural accelerations at a constant rate and stores them in an on-board buffer which can be accessed in-order through an SPI interface.

5.2. Accuracy vs. first-order hyper-parameters

As introduced in Section 2, to study the impact of the eLSNNs hyper-parameters and flavours on the accuracy, we conducted an exhaustive search. We evaluated the accuracy as the Matthews Correlation Coefficient (MCC) on the test set, which includes all the samples related to vehicle passages on the bride section during a randomly chosen day before and after the scheduled maintenance intervention.

In Figure 9, we report on the y-axis the network accuracy measured as MCC. We limited the plot to those hyper-parameters configurations leading to eLSNNs with an MCC ≥ 0.6. This corresponds to an accuracy of 0.77. Moreover, in the same figure, we draw the line at the MCC $= 0.75$. We consider all the eLSNNs configurations achieving an MCC higher than 0.75 as acceptable or "good" configurations. This threshold corresponds to accuracy above 0.88.

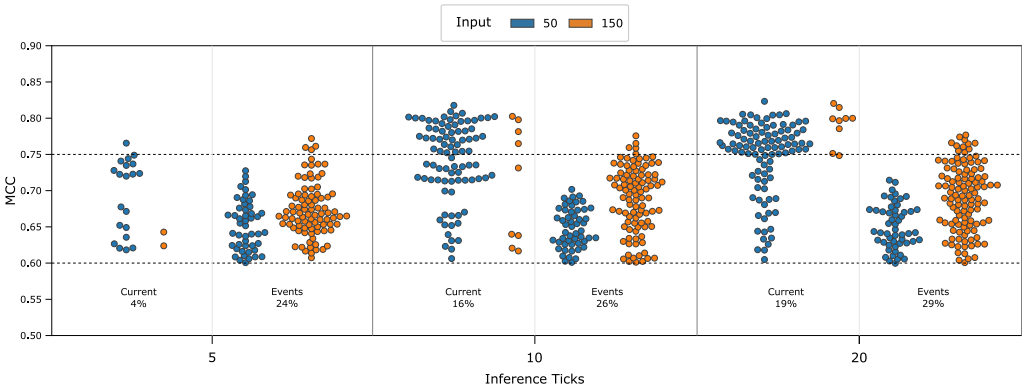

Figure 9. Inference ticks and number of inputs impact on MCC using the Current Input and the Event Input. Each point is a configuration of LSNN, which is characterised by its hyperparameters.

On the x-axis, we report three sets of plots. Each set corresponds to different inference ticks ($t_{inf} = 5, 10, 20$). Inside each set, the left plot refers to the current-driven eLSNN, and the right plot refers to the event-driven eLSNN. Inside each of these plots, we report the eLSNN configurations achieving a MCC > 0.6. The percentage of these configurations on the total evaluated (1728 for each eLSNN flavour) is reported in the text on each plot's bottom. The MCC accuracy of each configuration is reported with red bins for an input number of 50 and with blue bins for an input number of 150 bins. In the SHM application, the input number (I) corresponds to the magnitude of the spectral components of the FFT. It thus is equal to the double of the accelerations samples, which needs to be read by the sensor to compute an eLSNN inference.

The combination of these parameters (t_{inf}&I) constitutes the first-order hyper-parameter that, as we will see in the next section, impact the inference execution time of the eLSNNs and then their energy consumption. From the figure, we can notice that for the current-driven eLSNNs, the largest number of acceptable configurations is achieved for an input number (I) of 50. Which also corresponds to the lower complexity of the eLSNN computations (see Section4). Differently, for the event-driven eLSNNs, acceptable configurations can be achieved only with an input number (I) of 150. As we will see in Section5.4, this has a severe drawback on the pre-processing cost for this eLSNN flavour. It is also worth to note that for the current-driven eLSNNs the accuracy improves with larger inference ticks, having any acceptable configurations with $I = 50$ and $t_{inf} = 5$ but several with $I = 50$ and $t_{inf} = 10, 20$. However, this is not the case for the event-driven eLSNNs for which their MCC does not improve significantly with progressive t_{inf} increases. It is now interesting to evaluate these parameters' impact in terms of eLSNN inference execution cycles.

5.3. Execution time vs. activity-factors

As described in Section 4, the optimised eLSNN algorithm we propose in this paper leverages the sparse nature of the spikes for saving computations. If a neuron does not receive a spike, it does not trigger accumulation in the membrane potential. This means that the computational burden of the eLSNN algorithm depends on the average number of non-zero elements for each inference tick in the recursive/hidden layers. This is true for both current-driven and event-driven eLSNNs. For the event-driven eLSNN only, also the input layer must be considered in this computation. To understand the relevance of these effects on the total eLSNN inference time, we have to analyse different input samples. Each sample corresponds to the FFT coefficients of the accelerations read by the MCU during a passage of a vehicle. More specifically, the number of spikes corresponding to non-zero elements in the recurrent layer during an eLSNN inference (denoted with z) depends on the specific input sample and network hyper-parameters. Differently from the recurrent/hidden layer activity (z), the input activity factors vary

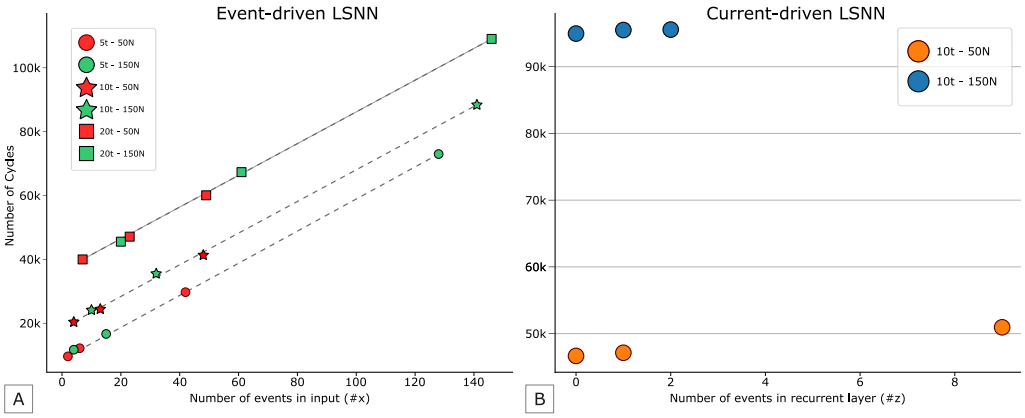

Figure 10. A) Cycle count for the event-driven eLSNN for the best configurations (input number with different colours, and inference ticks with different markers) computed on minimal, median and maximal spike activities on the input-layer (x). B) Cycle count for the current-driven eLSNN computed on minimal, median and maximal spike activities on the recurrent/hidden-layer (z). Different colour represent different number of inputs configuration for the eLSNN. All networks have an 10 inference ticks.

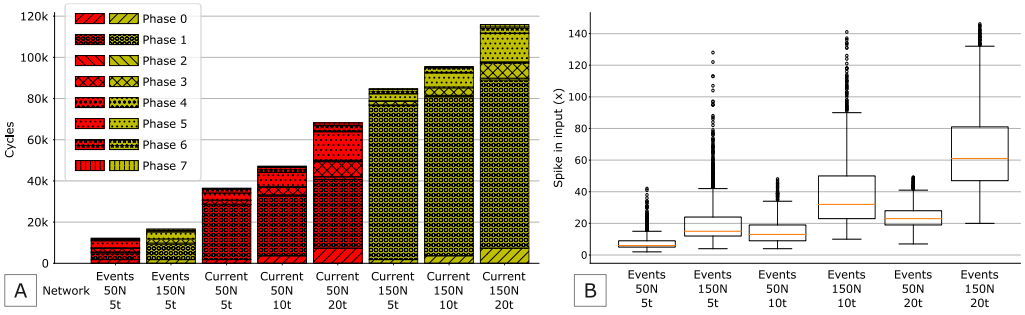

Figure 11. A) cycle count breakdown for the event-driven (event) and current-driven (current) eLSNN best configurations (input number - inference ticks) computed on median activity conditions. B) Distribution of the spikes in input for the event-driven eLSNN with different configurations (input number - inference ticks).

between the eLSNN flavour. For the current-driven eLSNNs, the input activity depends only on the input number (I). In contrast, for the event-driven eLSNNs, the input layer activity depends on the input spikes and non-zero elements in the coded inputs for all the ticks (x). x depends on the input sample and input number (I) as it is a property of the spiking input encoding.

Figure 10.B reports on the y-axis the total number of cycles needed by the eLSNN to complete one inference computation on a given input sample. This number of cycles accounts only for the eLSNN computation after the pre-processing step. On the x-axis, we report the value of z, which corresponds to the number of spikes/events/non-zero elements in the recurrent/hidden layer integrated into all the inference ticks (t_{inf}).

The different colours refer to the two different networks selected among the many with acceptable performance. For each type of network, we selected the ones with a number of inputs (I) leading to the largest (z) variation among the input samples in the test set.

Then, we plot three values of z chosen for each network corresponding to the minimum, the maximum, and the median sample. From the plot, we can notice that the impact of the z is negligible with respect to the total execution time of the eLSNN inference. Differently, the inference time for current-driven eLSNN halves when reducing the input number I from 150 to 50. As the impact of z is negligible, we can ignore its effect in the following plots.

Figure 11.B reports on the y-axis the distribution of the non-zero elements in the input layer for each input sample in the test set for event-driven eLSNNs. The distribution depends on the input element and the encoding, which depends on the first-order hyper-parameters, namely the inference ticks (t_{inf}) and the number of inputs (I). In the y-axis, we report the different configurations of the first-order hyper-parameters. From the plot, we can see that the number of events/spikes/non-zero elements in the input layer (x) is more significant and has higher variability than the same quantity in the recurrent/hidden layer (Figure10.B). Moreover, the number of input events/spikes/not-zero elements increases both in average and standard deviation with the increase of the input number and inference ticks.

Figure 10.A reports for each of the different configurations of the first-order hyper-parameters and for the input sample corresponding to the minimum, maximum and median values of x the execution time of the event-driven eLSNN. We can see that both the first-order hyper-parameters vary the execution time. The execution time increases linearly with the inference ticks and with x. It is interesting to notice that for the event-driven eLSNNs, the input number (I) does not increase the execution time directly as with the current-driven eLSNNs but indirectly. Indeed, a larger input number (I) increases the execution time proportionally to the increase of x. Which eLSNN flavour and configuration should thus be preferred?

5.4. Event-driven vs. current-driven

This section concludes the eLSNN study by comparing the performance of the most performing and energy-efficient current-driven, and event-driven eLSNN averaged on the test set. This is obtained by evaluating the candidate eLSNNs in the median sample with respect to both x and z.

Figure 11.A reports on the same plot the breakdown of the total number of cycles taken by the median inference time of both the current-driven and event-driven eLSNNs. For the current-driven networks, all the first-order hyper-parameters combinations are reported, while for the event-driven eLSNNs, we report only the configurations with inference ticks equal to 5, which corresponds to the most energy-efficient networks. By comparing the different networks, we can notice that the two most performing eLSNNs which achieves MCC ≥ 0.75 are the configuration $I = 150, t_{inf} = 5$ for the event-driven eLSNN and $I = 50, t_{inf} = 5$ for the current-driven eLSNN. The event-driven eLSNN requires less than half of the cycles of the current-driven eLSNN. We can conclude that event-driven eLSNN is significantly more efficient ($> 50\%$) than the current-driven eLSNN. It must be noted that this conclusion does not account for the pre-processing of the input sample (affecting the input number) which is more significant for the selected event-driven eLSNN. It is interesting to notice that the event-driven eLSNN configured with $I = 50, t_{inf} = 5$ is the most efficient one, but its MCC is lower (0.72) than the accuracy threshold of MCC ≥ 0.75. Moreover, in the Figure 11.A we report with different patterns the number of cycles needed to perform the computational steps described in 4.

- Phase 0: Compute of $\boldsymbol{v}^{(\alpha)}$
- Phase 1: Compute of $\boldsymbol{v}^{(I)}$
- Phase 2: Compute of $\boldsymbol{v}^{(H)}$
- Phase 3: Compute of $\boldsymbol{a}^{(\xi)}$
- Phase 4: Compute of $\boldsymbol{a}^{(th)}$ and $\boldsymbol{a}^{(z)}$
- Phase 5: Compute of $\boldsymbol{z}$ and insert items in $\check{\boldsymbol{z}}$
- Phase 6: Compute of $\boldsymbol{y}$
- Phase 7: Insert items in $\check{\boldsymbol{x}}$ for next iteration.

For the current-driven eLSNN the Phase 1 dominates the computational time since it must compute the $\boldsymbol{v}^{\boldsymbol{in}}$ vector by means of a complete matrix-vector multiplication. Phase 0 and Phase 3 involves N multiplications, and at increasing t_{inf} their execution times become increasingly evident.

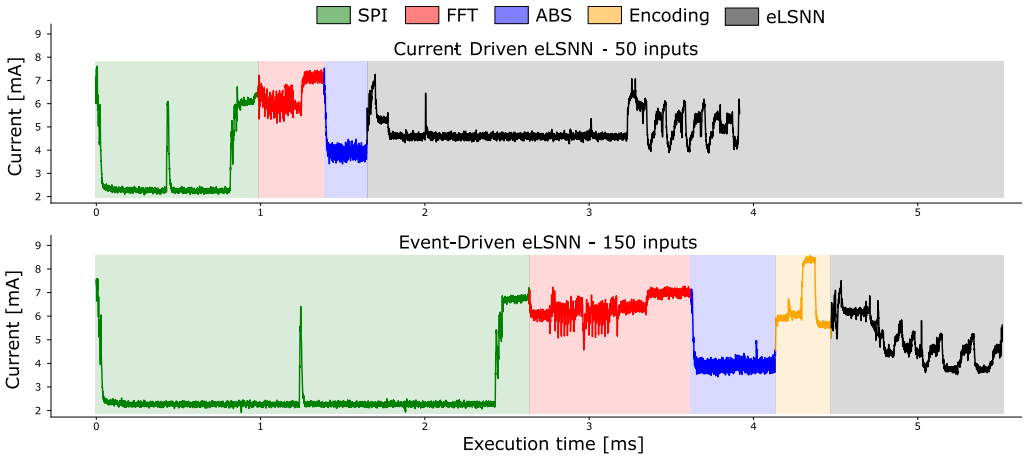

Figure 12. SHM application current consumption patterns for the best eLSNN networks working with current (top) and event (bottom) inputs. To better highlight the different stages of the application, the waveforms were obtained with the MCU clock slowed down to 16 MHz, while the SPI clock was 2 MHz.

Stage	Current-Driven				
	Time [μs]	Cycles [#]	Current [mA]	Power [mW]	Energy [μJ]
SPI	542	-	5.01	16.53	8.96
FFT	41	6971	38.76	127.91	5.24
ABS	26	4284	36.58	120.71	3.14
Encoding	-	-	-	-	-
eLSNN	215	36036	40.26	132.86	28.56
Total	824	47291	-	-	45.90
Mean	-	-	16.88	55.70	-

Table 3: Power report of the SHM application featuring the best eLSNNs tested. The SPI master transfers data with a serial clock of 8 MHz. The SPI stage includes: i) The SPI data transfer, ii) the time required by the system-on-chip to reconfigure the clock to 168 MHz when it wakes-up from SLEEP mode and iii) the conversion of sensor data from raw integers to floating-point.

Figure 12 shows the MCU's current consumption when executing the entire processing steps needed to read the sensor values through the SPI interface, pre-process the sample (FFT + ABS), and compute the eLSNN kernel. We report the current-driven eLSNN in its most efficient configuration on the left plot, and on the right plot, we report the event-driven eLSNN in its most efficient configuration. Even if the event-driven eLSNN kernel costs halves of the cycles than the current-driven eLSNN, it requires three times more sensor's readings to compute an inference for an input sample. This increases the cycles needed to read the sensor's data in SPI and compute the FFT and abs. Moreover, the event-driven eLSNN requires an additional pre-processing step consisting of the coding of the spectral components in spikes/events described in Section 3.

As described in section 4.5 the version of the network with current input needs a matrix-vector multiplication for each input tick $t_{inp} = t_{inf}/t_{exp}$. In this use case, we have only one input tick and only one matmul is executed, but it is enough to increase the number of cycles considerably. In Figure 12.A it is possible to appreciate this phase because it is visible for a long period (around milliseconds 2 and 3) not present in figure 12.B.

Tables 3 and 4 summarises all these effects. We can notice that even if the event-driven eLSNN kernel takes 54% fewer cycles than the current-driven eLSNN network, the total computation time is 1.51x longer for the event-driven eLSNN. This is primarily

Stage	Event-Driven				
	Time [μs]	Cycles [#]	Current [mA]	Power [mW]	Energy [μJ]
SPI	949	-	4.33	14.29	13.56
FFT	101	16968	38.17	125.96	12.72
ABS	50	8316	35.97	118.70	5.94
Encoding	32	5459	42.65	140.75	4.50
eLSNN	99	16631	38.20	126.06	12.48
Total	1231	47374	-	-	49.20
Mean	-	-	12.11	39.97	-

Table 4: Power report of the SHM application featuring the best eLSNNs tested. The SPI master transfers data with a serial clock of 8 MHz. The SPI stage includes: i) The SPI data transfer, ii) the time required by the system-on-chip to reconfigure the clock to 168 MHz when it wakes-up from SLEEP mode and iii) the conversion of sensor data from raw integers to floating-point.

due to the SPI transfer, which is 2x longer than for the current-driven eLSNN. Even with this extra execution time, the total time for computing the network matches the real-time requirements of the SHM application. Due to the lower power cost of the SPI transfer (w. DMA), however, the energy-consumption for the eLSNN flavours (event-driven and current-driven inputs) is comparable and in the range of 46 - 49 μJ.

In future works, we will explore temporal coding techniques and migrate the coding task directly in the MEMS sensor and in the time domain. These on-sensor coding techniques will remove the FFT cost and the amount of data to be read from the MEMS sensors, achieving further energy reduction in the sensor node.

6. Conclusion

In this paper, we presented an optimised implementation of a recurrent Spiking Neural Network on embedded microcontrollers for damage detection in Structural Health Monitoring applications. We studied the feasibility of spiking based processing and the trade-offs involved in using an event-based input. Thanks to this implementation, called eLSNN, we were able to design, implement and characterise an SNN-based monitoring system, where the computation is performed near to the sensor node. We described the optimisations we performed with respect to a naive LSNN algorithm present in the state-of-art to reduce computation cycles and improve energy efficiency. We also studied two alternative encodings of the input, showing how they impact performance and energy. We highlight the trade-offs between eLSNN execution and data transfer costs that have to be explored to select the best energy/performance configurations. Results of accuracy obtained on a real use case demonstrated that LSNN is a viable solution for damage detection in SHM, having high accuracy ($MCC \geq 0.75$) and low cycle and energy overheads if compared with the data transfer costs. Also, the results highlight that moving the spike-coding directly in the sensor can lead to even more energy-efficient implementation. The eLSNN working with input spikes is more energy-efficient than the one using real (continuous) values. The promising results in this work open the way to the implementation of SNN-based processing on edge. Moreover, when compared with a state-of-art work leveraging an autoencoder (ANN) on the same dataset [7], the proposed eLSSN approach shows lower complexity (from 32 000 MAC operations to 15 750 sums and 750 multiplications).

Future work will be devoted to evaluate and compare event-based SNN approaches against other SHM datasets and comparing extensively with ANN algorithms.

Acknowledgment

This work was supported by INSIST Project (PON Ricerca e Innovazione 2014-2020).

References

1. Smarsly, K.; Dragos, K.; Wiggenbrock, J. Machine learning techniques for structural health monitoring. Proceedings of the 8th European workshop on structural health monitoring (EWSHM 2016), Bilbao, Spain, 2016, pp. 5–8.
2. Burrello, A.; Marchioni, A.; Brunelli, D.; others. Embedded Streaming Principal Components Analysis for Network Load Reduction in Structural Health Monitoring. *IEEE Internet of Things Journal* **2021**, *8*, 4433–4447. doi:10.1109/jiot.2020.3027102.
3. Riziotis, C.; Testoni, N.; Aguzzi, C.; others. A Sensor Network with Embedded Data Processing and Data-to-Cloud Capabilities for Vibration-Based Real-Time SHM. *Journal of Sensors* **2018**, *2018*, 2107679. doi:10.1155/2018/2107679.
4. Foundation, T. TinyML. https://www.tinyml.org, 2021.
5. Microelectronics, S. STM32 solutions for artificial neural networks. https://www.st.com/content/st_com/en/ecosystems/stm32-ann.html, 2021.
6. Meyer, M.; Farei-Campagna, T.; Pasztor, A.; others. Event-Triggered Natural Hazard Monitoring with Convolutional Neural Networks on the Edge. Proceedings of the 18th International Conference on Information Processing in Sensor Networks; Association for Computing Machinery: New York, NY, USA, 2019; Ipsn '19, p. 73–84. doi:10.1145/3302506.3310390.
7. Moallemi, A.; Burrello, A.; Brunelli, D.; others. Model-based vs. Data-driven Approaches for Anomaly Detection in Structural Health Monitoring: a Case Study. 2021 IEEE International Instrumentation and Measurement Technology Conference (I2MTC). Ieee, 2021, pp. 1–6.
8. Maass, W. Networks of spiking neurons: The third generation of neural network models. *Neural Networks* **1997**, *10*, 1659–1671. doi:https://doi.org/10.1016/S0893-6080(97)00011-7.
9. Indiveri, G.; Horiuchi, T. Frontiers in Neuromorphic Engineering. *Frontiers in Neuroscience* **2011**, *5*, 118. doi:10.3389/fnins.2011.00118.
10. Roy, K.; Jaiswal, A.; Panda, P. Towards spike-based machine intelligence with neuromorphic computing. *Nature* **2019**, *575*, 607–617. doi:10.1038/s41586-019-1677-2.
11. Deng, L.; Wu, Y.; Hu, X.; others. Rethinking the performance comparison between SNNS and ANNS. *Neural Networks* **2020**, *121*, 294–307. doi:https://doi.org/10.1016/j.neunet.2019.09.005.
12. Lamba, S.; Lamba, R. Spiking Neural Networks Vs Convolutional Neural Networks for Supervised Learning. 2019 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS), 2019, pp. 15–19.
13. Neves, A.C.; González, I.; Leander, J.; others. Structural health monitoring of bridges: a model-free ANN-based approach to damage detection. *Journal of Civil Structural Health Monitoring* **2017**, *7*, 689–702. doi:10.1007/s13349-017-0252-5.
14. Seventekidis, P.; Giagopoulos, D.; Arailopoulos, A.; others. Structural Health Monitoring using deep learning with optimal finite element model generated data. *Mechanical Systems and Signal Processing* **2020**, *145*.
15. Avci, O.; Abdeljaber, O.; Kiranyaz, S.; Inman, D. Structural damage detection in real time: implementation of 1D convolutional neural networks for SHM applications. In *Structural Health Monitoring & Damage Detection, Volume 7*; Springer, 2017; pp. 49–54.
16. Yang, L.; Fu, C.; Li, Y.; others. Survey and study on intelligent monitoring and health management for large civil structure. *International Journal of Intelligent Robotics and Applications* **2019**, *3*, 239–254. doi:10.1007/s41315-019-00079-2.
17. Pang, L.; Liu, J.; Harkin, J.; others. Case Study, Spiking Neural Network Hardware System for Structural Health Monitoring. *Sensors* **2020**, *20*. doi:10.3390/s20185126.
18. Madrenas, J.; Zapata, M.; Fernández, D.; others. Towards Efficient and Adaptive Cyber Physical Spiking Neural Integrated Systems. 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2020, pp. 1–4. doi:10.1109/icecs49266.2020.9294982.
19. Bellec, G.; Scherr, F.; Subramoney, A.; others. A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications* **2020**, *11*, 1–15.
20. Stromatias, E.; Soto, M.; Serrano-Gotarredona, T.; others. An Event-Driven Classifier for Spiking Neural Networks Fed with Synthetic or Dynamic Vision Sensor Data. *Frontiers in Neuroscience* **2017**, *11*, 350. doi:10.3389/fnins.2017.00350.
21. Burrello, A.; Marchioni, A.; Brunelli, D.; others. Embedded Streaming Principal Components Analysis for Network Load Reduction in Structural Health Monitoring. *IEEE Internet of Things Journal* **2020**.
22. Ltd., R.P.T. Raspberry Pi Compute Module 3+ DATASHEET. https://www.raspberrypi.org/documentation/hardware/computemodule/datasheets/rpi_DATA_CM3plus_1p0.pdf, 2019.
23. HiveMQ. HiveMQ Documentation V4.5. https://www.hivemq.com/docs/hivemq/4.5/user-guide/introduction.html, 2019.
24. Tokognon, C.A.; Gao, B.; Tian, G.Y.; others. Structural health monitoring framework based on Internet of Things: A survey. *IEEE Internet of Things Journal* **2017**, *4*, 619–635.
25. Moradi, S.; Qiao, N.; Stefanini, F.; others. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE transactions on biomedical circuits and systems* **2017**, *12*, 106–122.
26. Schemmel, J.; Fieres, J.; Meier, K. Wafer-scale integration of analog neural networks. 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence). Ieee, 2008, pp. 431–438.
27. Benjamin, B.V.; Gao, P.; McQuinn, E.; others. Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations. *Proceedings of the IEEE* **2014**, *102*, 699–716.

28. Davies, M.; Srinivasa, N.; Lin, T.H.; others. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro* **2018**, *38*, 82–99.
29. DeBole, M.V.; Taba, B.; Amir, A.; others. TrueNorth: Accelerating From Zero to 64 Million Neurons in 10 Years. *Computer* **2019**, *52*, 20–29. doi:10.1109/mc.2019.2903009.
30. Kim, J.; Caire, G.; Molisch, A.F. Quality-aware streaming and scheduling for device-to-device video delivery. *IEEE/ACM Transactions on Networking* **2015**, *24*, 2319–2331.
31. Ponulak, F.; Kasinski, A. Introduction to spiking neural networks: Information processing, learning and applications. *Acta neurobiologiae experimentalis* **2011**, *71*, 409–433.
32. Barchi, F.; Zanatta, L.; Parisi, E.; others. An Automatic Battery Recharge and Condition Monitoring System for Autonomous Drones. 2021 IEEE International Workshop on Metrology for Industry 4.0 IoT, 2021.
33. Instruments, N. PXI Systems. https://www.ni.com/it-it/shop/pxi.html, 2021.