UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Elettronica

# Online Learning of Oil Leaks Anomalies in Wind Turbines with Block Based Neural Network using Binary Reservoirs

# Apprendimento Online di Anomalie Relative a Perdite d'Olio in Turbine Eoliche Tramite Rete Neurale con Processamento a Blocchi e Reservoir Binari

Advisor:
**Prof. Claudio Turchetti**

Coadvisor:
**Dr. Laura Falaschetti**
**Ing. Danilo Pietro Pau**

Candidate:
**Matteo Cardoni**

Academic Year 2020-2021

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Elettronica
Via Brecce Bianche – 60131 Ancona (AN), Italy

# Abstract

Wind power is one of the most used renewable energy technologies. Human inspections in the energy generator are expensive and require highly specialized personnel as turbines are located at hundreds of meters of altitudes, surrounded by harsh environments. Therefore, to reduce un-needed intervenes or periodic assessments, accurate and intelligently automated anomaly detection will play an important role. The focus of this work is to design a deeply quantized anomaly detector of oil leaks which may happen at the junction between the turbine high speed shaft and the external bracket of the power generator. Here it is proposed a Block Based Binary Shallow Echo State Network (BBS-ESN) architecture, belonging to reservoir computing (RC) category, that is constituted by two reservoir layers. Furthermore, BBS-ESN uses binary images block based processing and block based online training. This is done using fixed and minimal computational complexity, permitting to achieve low power consumption and deployability on an off the shelf micro-controller (MCU). This has been achieved through binarization of the images and up to 1-bit quantization of the network weights and activations.

3D rendering has been used to generate a novel dataset of photo realistic images similar to those potentially acquired by image sensors on the field while monitoring the junction, with and without oil leaks. Images have been binarized and image morphing has been used in order to simulate oil leak propagation. Binary random noise, different from image to image, has been added to the sequences obtained with morphing to simulate various percentages of defective photo-transistors, with time unrelated malfunctioning.

Best achieved results in the less challenging conditions (i.e. without addition of binary noise) are an accurate identification of anomalies, with 0% of false negatives and 0% of false positives. Best achieved results in the most challenging conditions (i.e. with an addition of binary noise in 10% of the pixels) are of 4.1% of false positives and 14.6% of false negatives. The computational cost per image inference is in the order of $7.8 \cdot 10^6$ for binary multiplications, $3.9 \cdot 10^6$ for *int8* sums and $3.9 \cdot 10^6$ for *int16* sums. The used RAM and Flash memories are respectively of 129.2 KBytes and 16 KBytes. The inference execution time estimated using a STM32H743ZI2 MCU running at 480MHz is about 12.06 ms per image.

The extra computational costs for the BBS-ESN initial online training are in the order of $3.1 \cdot 10^7$ for binary multiplications, $1.1 \cdot 10^8$ for *fp32* multiplications, $3.1 \cdot 10^7$ for *int8* additions and $2.6 \cdot 10^5$ for *fp32* divisions. The estimated required RAM is of 2,176 KBytes while the estimated required time, using the same MCU, is 1.232 s. The anomlay decision process, after the BBS-ESN training, sets a threshold to separate the anomalous images from the normal ones in 1.206 s.

On the basis of these results it can be concluded that BBS-ESN is feasible on off-the-shelf 32bit MCUs. Moreover the solution is also scalable in the number of cameras to be deployed, permitting to achieve accurate and fast oil leak detections

from different view points.

## Sommario

Le turbine eoliche sono una delle tecnologie più utilizzate per lo sfruttamento di energie rinnovabili. L'ispezione umana all'interno delle cabine delle turbine eoliche è costosa, poiché richiede personale altamente specializzato e poiché i generatori possono trovarsi a centinaia di metri di altitudine, in ambienti ostili. Per questo motivo rivelatori di anomalie automatici possono ricoprire un importante ruolo nella riduzione di interventi specializzati inutili o di controlli periodici. L'obiettivo di questo studio è la progettazione di un rilevatore di anomalie, che usi pesi fortemente quantizzati, per identificare le perdite d'olio che possono verificarsi alla giunzione tra l'albero ad alta velocità della turbina e l'involucro del generatore elettrico. Viene proposta una rete neurale artificiale della categoria Reservoir computing (RC) che usa due strati di reservoir, denominata Block Based Binary Shallow Echo State Network (BBS-ESN). La BBS-ESN processa immagini binarie in blocchi e si addestra usando blocchi di immagini. Questi calcoli sono effettuati con una complessità di calcolo bassa e fissa, permettendo di raggiungere un basso consumo di potenza e di implementare la rete su microcontrollori (MCU) disponibili in commercio. Questo è possibile a causa della intensiva quantizzazione dei pesi e delle attviazioni della rete, fino a quantizzare con un solo bit.

Il rendering 3D è stato usato per generare un dataset di immagini fotorealistiche appositamente creato. Queste immagini sono simili a quelle potenzialmente acquisite dai sensori di immagine sul campo durante il monitoraggio della giunzione. Le immagini sono state binarizzate e il morphing per immagini è stato usato per simulare la propagazione delle macchie d'olio. Alle sequenze così ottenute è stato sommato rumore binario differente da immagine a immagine, così da simulare malfunzionamenti nei foto transistors non correlati nel tempo, in varie percentuali.

I risultati ottenuti nelle migliori condizioni, ovvero senza l'aggiunta di rumore alle sequenze, è un'accurata identificazione delle anomalie con lo 0% di falsi negativi e lo 0% di falsi positivi. Per quanto riguarda i risultati ottenuti nelle condizioni più sfidanti, cioè usando sequenze con rumore aggiunto al 10% dei pixel, questi sono di 4.1% di falsi positivi e di 14.6% di falsi negativi. I costi computazionali per l'inferenza di una immagine sono nell'ordine di $7.8 \cdot 10^6$ per le moltiplicazioni binarie, $3.9 \cdot 10^6$ per le somme *int8* e $3.9 \cdot 10^6$ per le somme *int16*. Le memore RAM e Flash usate sono rispettivamente di 129.2 KBytes e 16 KBytes. Il tempo di esecuzione stimato per l'inferenza, usando il microcontrollore STM32H743ZI2 con frequenza di clock 480 MHz, è di circa 12.06 ms per immagine.

I costi computazionali aggiuntivi per l'addestramento online iniziale della BBS-ESN sono nell'ordine di $3.1 \cdot 10^7$ per le moltiplicazioni binarie, $1.1 \cdot 10^8$ per le moltiplicazioni *fp32*, $3.1 \cdot 10^7$ per le somme *int8* e $2.5 \cdot 10^5$ per le divisioni *fp32*. La stima

della RAM richiesta per l'addestramento online è di a 2,176 KByes mentre il tempo richiesto, usando lo stesso microcontrollore alla stessa frequenza di clock, è stimato a 1.232 s. Il processo di decisione per le anomalie, dopo l'addestranento della BBS-ESN, impone una soglia per separare le immagini anomale da quelle normali in circa 1.206 s.

Sulla base di questi risultati si può concludere che la BBS-ESN è implementabile su un microcontrollore a 32bit disponibile in commenrcio. Inoltre, la soluzione è scalabile nel numero di sensori di immagine usati, permettendo di ottenere rilevazioni di anomalie accurate e veloci da punti di vista differenti.

# Contents

## List of Figures

# List of Tables

# 1 Introduction

The wind industry is in continuous growth. Global installed wind-generation capacity onshore and offshore has increased by a factor of almost 75 in the past two decades, increasing from 7.5 gigawatts (GW) in 1997 to some 564 GW by 2018, according to [1]. Provision of wind electricity doubled between 2009 and 2013, and in 2016 wind energy accounted for 16% of the electricity generated by renewables, as reported in [2]. Wind turbines are typically clustered together in wind farms, each of which can generate enough electricity to power thousands of homes. Wind farms are usually located on onshore on top of a mountain or in an otherwise windy place or, if offshore, in the sea in order to exploit natural winds.

By consequence, wind turbines inspection, monitoring, and repairing is a very complex and costly task performed in a harsh environment and represents a fast growing industry due to the high number of existing wind turbines that, approaching to an advanced age, require maintenance. As these wind turbines age, predictive maintenance is emerging as a fundamental requirement. Predictive maintenance differs from preventive maintenance because it relies on the actual condition of the equipment, rather than on the average or expected life statistics used to predict when maintenance will be necessary. Typically, Machine Learning approaches are adopted for the estimate of the actual condition of the system, including the capability to detect anomalies. In particular oil leaks detection in Wind Turbines is an interesting problem, since leaks from the power generator can provoke fires inside the turbine and because human prompt intervene is very difficult to happen, being both dangerous and very expensive. In fact, the access to the inside cabin of the generator is challenging as the turbines may be located offshore or in hard to access locations. Figure 1 is shown a simplified scheme of the internal components of a wind turbine [3]. The the junction where the anomalies are to be detected, between the high speed shaft and the external bracket of the power generator, is put in evidence.

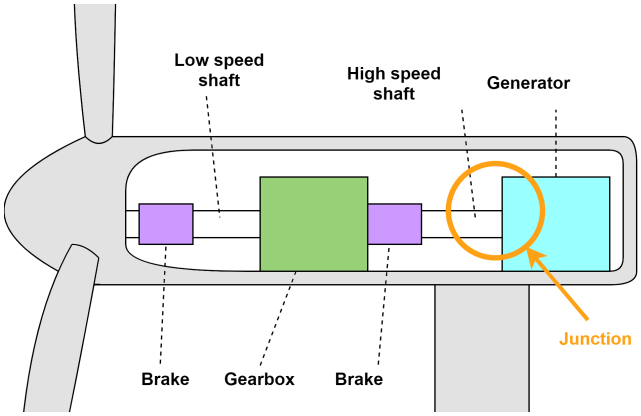


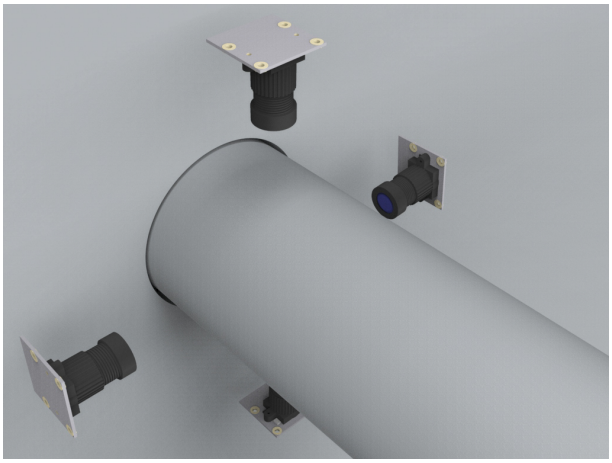

Figure 1: Scheme of the internal of a wind turbine. The orange circle indicates the region of interest for anomaly detection.

The inside of the cabin offers an interesting opportunity: it can be fully controlled using image cameras since the internal environment is un-changing over the time and no lighting changes can happen. In fact, the external weather conditions cannot affect the inside. This suggests that the deployment of image cameras to monitor them is much less affected by illumination changes than cameras operating in other open un-controlled environments, such as autonomous driving. Therefore, cameras can be used to perform oil leaks detection in a kind of laboratory controlled working conditions, allowing the oil leak to be the only possible change to be detected. Once any anomaly will be detected, it will be promptly communicated to the maintenance service department, so that a specialized operator intervention will be programmed. The proposed system is conceived in such a way that image sensors are framing the junction between the shaft and the bracket of the power generator, where oil leaks would be located in case of anomaly. The junction needs to be framed with a 360° field of view. For this reason, at least 3 or 4 cameras would be needed (the zones framed by the camera can also overlap), depending on their field of view. Figure 2 shows a simplified scheme, using 4 cameras to frame the junction.

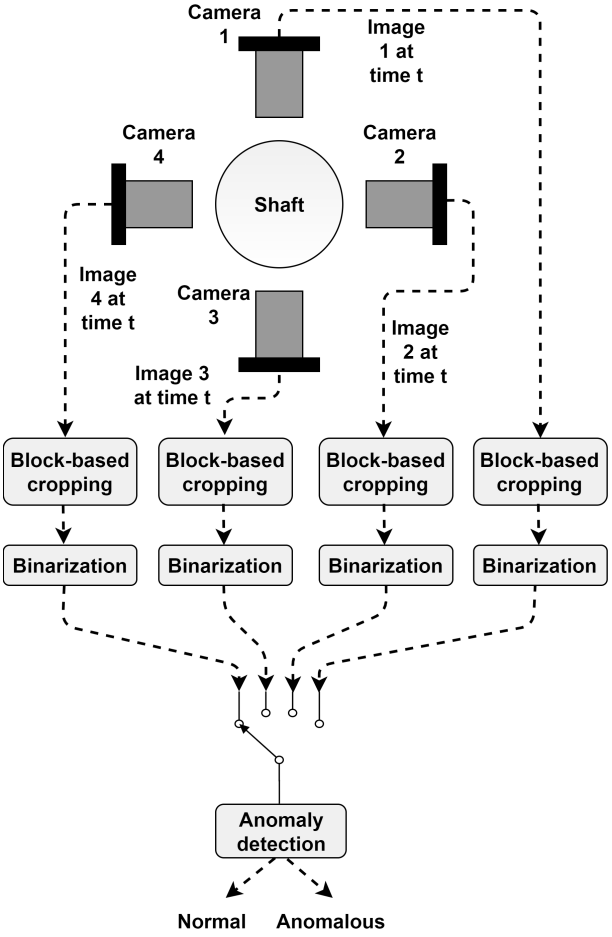
Such a system architecture requires to design a machine learning based anomaly detector per each camera, to detect anomalies 360°.

Inside the images obtained from each camera, a precisely defined zone of interest is defined, limiting the search area of the leaks, with also the advantage of reducing the processing complexity. Since, at the best of our knowledge, no public dataset about this problem exists in the state of the art, images have been produced through 3D graphics photo-realistic rendering, reproducing scenarios with and without anomalies and from different view points, i.e. camera positions. The challenge we addressed consists of designing a system that can learn the normality in an online modality, with low and fixed computational complexity as well as fixed memory storage during its functioning. Thus, we investigated a system that can possibly run with low power consumption on a cheap and off-the-shelf micro-controller unit (MCU) which, in a preferred embodiment, is attached to all those image sensors instantiated into the system to achieve close-to-sensor machine learning implementation.

The constraint of fixed complexity learning brought us to the choice of an Echo State Network, which belongs to the category of Reservoir Computing category, as will be specified in the section 3. Low complexity requirements are not only achieved by designing a neural network with parsimonious model size, but also binarizing the captured images before being processed by the neural network. This is also possible due to the controlled lighting and overall working conditions inside the turbine generator cabin. The binarization of the images, in turn, creates another opportunity: to further reduce the complexity of the network by deeply quantizing and, at most, binarizing its weights, including the reservoir layers ones and the activations. In order to aggressively reduce the memory occupation and improve pipelined processing of the system respect to a network working on the entire image,



(a) The cameras framing the junction




(b) Block diagram of the cameras framing the junction, being synchronized in time

Figure 2: Exemplary installation with 4 cameras framing the junction where anomalies can happen.

the framed zones of interest feeding the neural network are furtherly decomposed into non overlapping blocks. This mimics a sequential scan in, for example, lexicographic order, or in any other as the sensor technology permits. Finally, we designed the system with the aim to make it robust to defective photo-transistors in the image sensors, in order to deal with the lifetime of the system. This, also, would permit to reduce the overall system maintenance. The sensor's photo-transistors may in fact be damaged by imperfect manifacturing process, aging or averse ambiental conditions, like temperature fluctuations. In summary the peculiarity of the proposed work is in the use of a deeply quantized (8 and 1 bit) block based ESN network with more than one reservoir layers, capable to perform online normality (oil leaks absence) learning with a cap on complexity. The system, also, uses images decomposed in a block based temporally processed sequence. It is followed by an special purpose anomaly decision process of the network outputs, named Evaluator. The proposed network has been named Block Based Binary Shallow Echo State Network, briefly BBS-ESN.

## 2 System architecture and associated requirements

In Figure 3 the block diagram of the system architecture is shown, from the images acquisition in one of the cameras to their classification as normal or anomalous. The outputs of the neural network are evaluated with a decision process, also implemented by the MCU, called Evaluator, that will be described in detail in section 7.

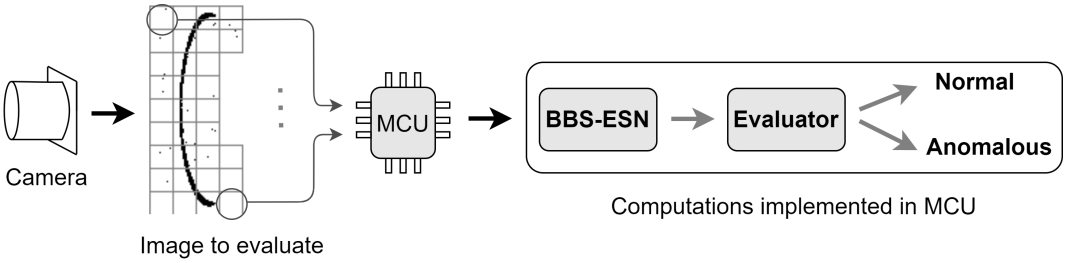



Figure 3: BBS-ESN netowrk with camera inputs, processing them as temporal sequences of blocks.

The system requirements, from which the architecture is derived, are listed as follows:

**Cameras:** a simplyfied diagram of the system architecture is shown in Figure 2b, where 360 degree monitoring of the junction is achieved, as exemplary embodiment, using 4 cameras with adequate field of view (as explained in section 4). The number of cameras to be used depends on their field view, so the system needs to be scalable in their number.

**Block Based Processing:** each camera outputs only a limited set of fixed position blocks, that are the ones that outline the zone of interest, that is the

junction. These images blocks are binarized, since the dark shape of leaks is the only relevant information to solve the problem. In fact, there is no need to process color information which were considered redundant and deceitful. The binarization adopted in this study is explained in subsubsection 4.4.3. The binarized blocks are used as input by the micro-controller unit (MCU) whose aim consists of detecting the presence of any oil leaks, implementing the presented machine learning solution.

**Low Complexity requirements:** The processing complexity that any MCU needs to support has to be minimized without compromising the accuracy of the solution. If that would be achieved, then the derived approach can be implemented on a low power micro controller, considering the problem of the low embedded RAM and FLASH memory available.

As a consequence, the neural network has to be carefully designed. Besides limiting the network weights number, complexity and memory reduction can be achieved with quantization, even binarization in some cases, of the weights and of the activations. To further decrease memory buffering complexity block-based processing is used, mimicking each blocks decomposed image in a kind of temporal sequence.

**Online learning:** since the system must not require pre-knowledge or pre-calibration, the system must be able to learn the normality condition on site. After the learning is concluded in a fixed amount of time, the system shall be able to detect anomalies with low number of false negatives (to avoid missing dangerous anomalies) and false positives (to avoid to trigger un-needed human intervene).

**Absence of real time constraints:** the possibility to implement the detection in an MCU, with small power consumption, is also made possible by the absence of real time constraints. Therefore, once the zones of interest of an image are processed by the micro controller, that end can trigger the image acquisition to happen. Another assumption that justifies the absence of real time operations necessity is that the human maintainer intervene time is order of magnitudes greater than the detection time.

# 3 Related work

## 3.1 Review of Echo State Networks and Deep Echo State Networks

Due to the problem and requirements discussed in section 2, inference and learning need to be performed in a fixed and predictable amount of time and memory. These needs represent a major limitation in the choice of NN topologies since most training methods converge to the optimal solution in an a-priori unknown amount

of computational and storage resources. Typical training procedure of NN is based on back-propagation using stochastic gradient descend using multiple epochs [4] by processing large batches of data.

To overcome these problems, the BBS-ESN uses Deep Echo State Networks (DeepESN briefly) [5, 6], that follow the principles of Echo State Networks (ESN briefly) [7, 8]. They both belong to the category of Reservoir Computing (RC), with the difference that DeepESN use more than one reservoir layer. In ESN and DeepESN hidden nodes values are randomly assigned and never updated, while the output weights layer are trained with a linear regression method [7], allowing an a-priori known computations, memory and time. These networks, also, are suited for temporal tasks.

In the field of artificial intelligence we can define a task as the learning of a function that relates the input $\boldsymbol{u}$ with the desired output $\boldsymbol{y}_{target}$, with the objective of minimizing the an error function $E(\boldsymbol{y}, \boldsymbol{y}_{target}(t))$. A temporal task is defined as the learning of a function $\boldsymbol{y}(t) = \boldsymbol{y}\big(\ldots, \boldsymbol{u}(t-1), \boldsymbol{u}(t)\big)$ in the discrete time domain $t = 0, \ldots, T$ [8].

### 3.1.1 Review of Echo State Networks

ESN use a layer that is called reservoir, composed by $N_n$ neuron units. The activation input vector is defined as $\boldsymbol{u}(t) \in \mathbb{R}^{N_u}$, the network state vector as $\boldsymbol{x}(t) \in \mathbb{R}^{N_n}$ and the output vector as $\boldsymbol{y}(t) \in \mathbb{R}^{N_y}$.

The weights matrices associated to the vectors are $\boldsymbol{W}_{in} \in \mathbb{R}^{N_n \times N_u}$ for $\boldsymbol{u}(t)$, $\boldsymbol{W} \in \mathbb{R}^{N_n \times N_n}$ for the internal connections and $\boldsymbol{W}_{out} \in \mathbb{R}^{N_y \times N_n}$ for the output.[7] In the situation that there is not a weights matrix that connects the output with the input (like the case in this study), we can write the network state at time $t$ as

$$\boldsymbol{x}(t) = f\big(\boldsymbol{W}_{in}\boldsymbol{u}(t) + \boldsymbol{W}\boldsymbol{x}(t-1)\big) \tag{1}$$

where $f$ is a non-linear function applied element-wise.

The output, in the situation where $\boldsymbol{W}_{out}$ is connected only to the network state, is calculated as:

$$\boldsymbol{y}(t) = \boldsymbol{W}_{out}\boldsymbol{x}(t) \tag{2}$$

The ESN scheme is represented in Figure 4.

A condition that is usually wanted in the matrix $\boldsymbol{W}$ is the *echo state property*. This condition states that the effect of a state $\boldsymbol{x}(t)$ and an input $\boldsymbol{u}(t)$ on a future state $\boldsymbol{x}(t+k)$ diminishes gradually with the succession of temporal instants. That is to say that the effects vanishes for $k \to \infty$. Such property is obtained multiplying the matrix $\boldsymbol{W}$ for a scaling coefficient such that the spectral radius $\rho(\boldsymbol{W})$, that is the greatest modulus of the eigenvalues of the matrix $\boldsymbol{W}$, satisfies $\rho(\boldsymbol{W}) < 1$.[8]

In this study, for the design of the quantized BBS-ESN, is assumed a less restrictive condition on $\rho(\boldsymbol{W})$. This condition is $\rho(\boldsymbol{W}) \leq 1$, as explained in [9]. The matrix

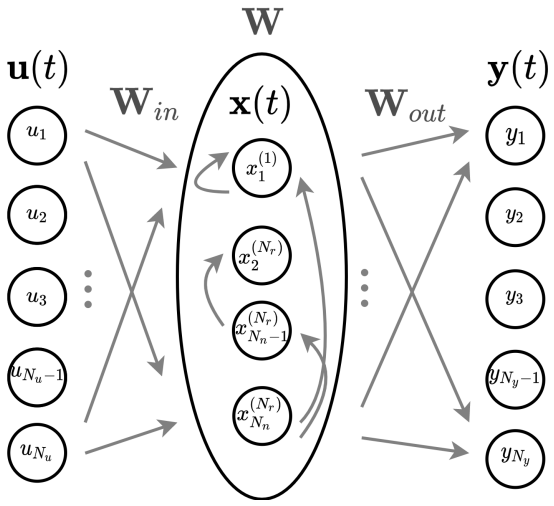



Figure 4: Scheme of an Echo State Network

$\boldsymbol{W}_{in}$ is usually set as dense, that is with all the weights different from 0 [8]. On the contrary, the matrix $\boldsymbol{W}$ is usually sparse (that is with some of the weights equal to 0), with a connectivity percentage of 20% or less (i.e. 20% or less of non-zero weights). [10].

### 3.1.2 Training of an Echo State Network

In an Echo State Network the only matrix that is subject to training is $\boldsymbol{W}_{out}$. The matrices $\boldsymbol{W}_{in}$ e $\boldsymbol{W}$ are instantiated with random values and kept fixed [7, 8, 9]. The reason for this is that the matrices for the state and for the output computation serve different purposes: $\boldsymbol{W}_{in}$ and $\boldsymbol{W}$ expand the input history $\boldsymbol{u}(t-1), \boldsymbol{u}(t), \ldots$ into a rich enough state space, while $\boldsymbol{W}_{out}$ combines the state in a vector $\boldsymbol{y}(t)$ more similar as possible to the target output vector $\boldsymbol{y}_{target}(t)$ [8].

The training of the readout matrix takes place in batch, grouping the states from $T$ preceding instants in

$$\Big[\boldsymbol{x}(1), \ldots, \boldsymbol{x}(T-1), \boldsymbol{x}(T)\Big] = \boldsymbol{X} \in \mathbb{R}^{N_n \times T} \tag{3}$$

and the correspondent readout for each temporal instant

$$\Big[\boldsymbol{y}_{target}(1), \ldots, \boldsymbol{y}_{target}(T-1), \boldsymbol{y}_{target}(T)\Big] = \boldsymbol{Y}_{target} \in \mathbb{R}^{N_y \times T} \tag{4}$$

With these vectors grouped, the training is based on the solution of the linear system

$$\boldsymbol{W}_{out}\boldsymbol{X} = \boldsymbol{Y}_{target}, \tag{5}$$

*for this reason the training can take place online.*

The solution could be found using a direct method, calculating the Moore-

Penrose pseudo-inverse matrix:

$$\boldsymbol{W}_{out} = \boldsymbol{Y}_{target}\boldsymbol{X}^{+}. \tag{6}$$

This method would be computationally expensive, so it preferable to reformulate the problem as:

$$\boldsymbol{W}_{out}\boldsymbol{X}\boldsymbol{X}^{T} = \boldsymbol{Y}_{target}\boldsymbol{X}^{T}. \tag{7}$$

Since $\boldsymbol{X}\boldsymbol{X}^{T}$ is square it can be inverted, leading to:

$$\boldsymbol{W}_{out} = \boldsymbol{Y}_{target}\boldsymbol{X}^{T}(\boldsymbol{X}\boldsymbol{X}^{T})^{-1}. \tag{8}$$

However, there is the possibility for the matrix $\boldsymbol{X}\boldsymbol{X}^{T}$ to be singular. The method used in the equation 8 can be modified with a regularization term to increase numerical stability, leading to the equation

$$\boldsymbol{W}_{out} = \boldsymbol{Y}_{target}\boldsymbol{X}^{T}(\boldsymbol{X}\boldsymbol{X}^{T} + \alpha^{2}\boldsymbol{I})^{-1}, \tag{9}$$

where $\boldsymbol{I} \in \mathbb{N}^{N_n \times N_n}$ is an identity matrix [8].

### 3.1.3 Review of Deep Echo State Networks

A Deep Echo State Network follows the principles of ESN, but is composed by more than one reservoir layers, connected in cascade.

The rationale behind the stacking of reservoir layers is the processing of each temporal information in a hierarchical fashion, that is through composition of multiple levels of recurrent units [6].

In a DeepESN the first reservoir receives in input at time $t$ the activation vector $\boldsymbol{u}(t)$, at the same way an ESN does. The reservoirs after the first receive as input the state of the reservoir preceding them at the same instant.

Therefore, each reservoir state is calculated as

$$\boldsymbol{x}^{(r)}(t) = f\Big(\boldsymbol{W}_{in}^{(r)}\boldsymbol{i}^{(r)}(t) + \boldsymbol{W}^{(r)}\boldsymbol{x}^{(r)}(t-1)\Big), \tag{10}$$

where the superscript $(r)$ refers to the elements of the $r^{th}$ reservoir, starting from $r = 1$ for the first. $\boldsymbol{i}^{(l)}(t)$ describes the input at the $r^{th}$ reservoir at the instant $t$, that is:

$$\boldsymbol{i}^{(r)}(t) = \begin{cases} \boldsymbol{u}(t) & \text{if} \quad r = 1 \\ \boldsymbol{x}^{(r-1)}(t) & \text{if} \quad r > 1 \end{cases} \tag{11}$$

[5]. For a DeepESN with a number of reservoirs $N_r$, composed each by a number $N_n$ of neurons, the network state at time $t$ is the vertical stacking of the states of each reservoir at the same time (taken as column vectors):

$$\boldsymbol{x}(t) = \Big[\boldsymbol{x}^{(1)}(t)\ \boldsymbol{x}^{(2)}(t)\ ...\ \boldsymbol{x}^{(N_r)}(t)\Big]^{T}, \tag{12}$$

with each reservoir state $\boldsymbol{x}^{(r)}(t) \in \mathbb{R}^{N_n \times 1}$. The output of the network is:

$$\boldsymbol{y}(t) = \boldsymbol{W}_{out} \left[ \boldsymbol{x}^{(1)}(t)\ \boldsymbol{x}^{(2)}(t)\ ...\ \boldsymbol{x}^{(N_r)}(t) \right]^T, \tag{13}$$

where $\boldsymbol{W_{out}} \in \mathbb{R}^{N_y \times N_r N_n}$ is the weights matrix connecting the units of every reservoir to the network output, or readout, and the readout is $\boldsymbol{y}(t) \in \mathbb{R}^{N_y \times 1}$. The DeepESN scheme is represented in Figure 5

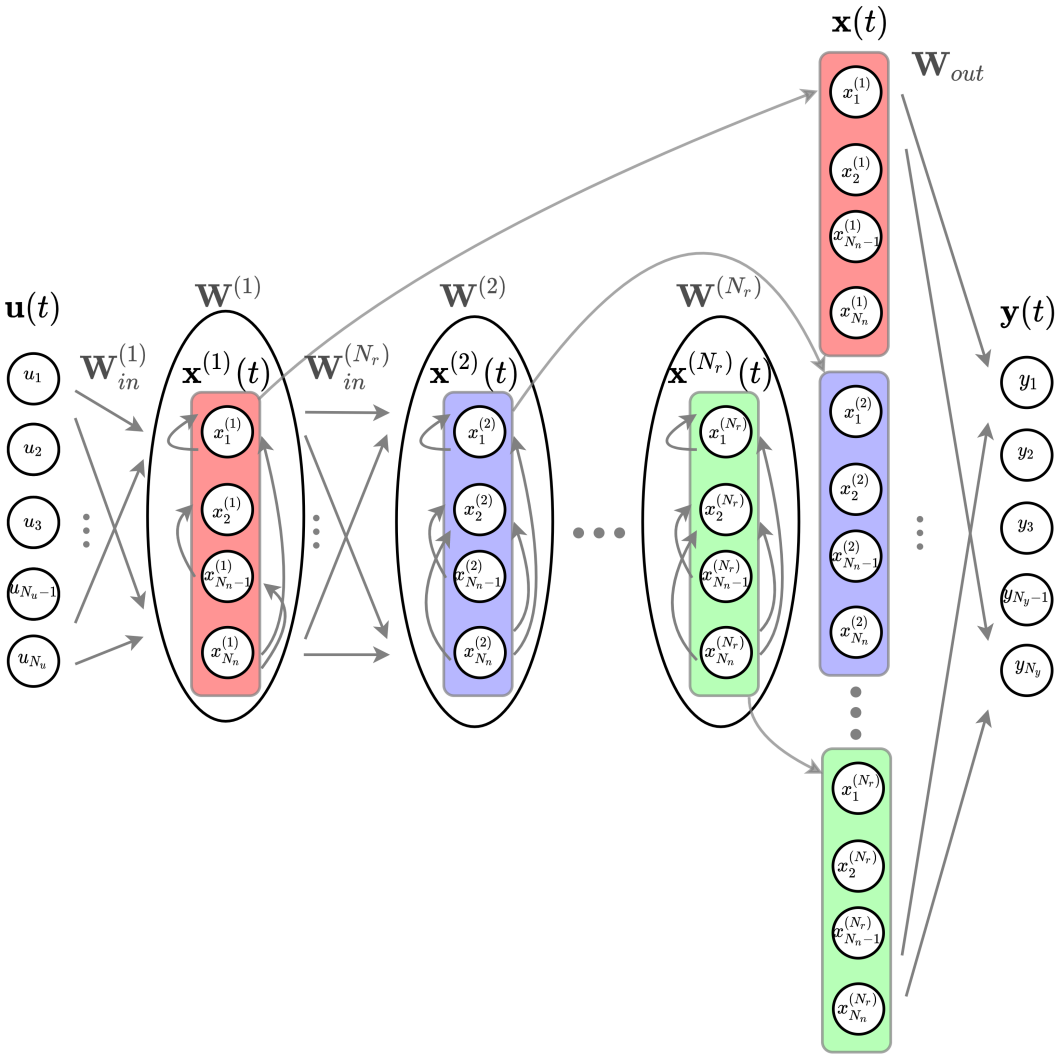



Figure 5: Scheme of a Deep Echo State Network

### 3.1.4 Training of Deep Echo State Network

Considering the state at time $t$ as Equation 12, the network training takes place composing the state matrix (Equation 3) with the states from Equation 12. This means that the state matrix, composed by the states of $T$ instants, will be composed

in the following way:

$$\begin{bmatrix} \boldsymbol{x}^{(1)}(1) & \dots & \boldsymbol{x}^{(1)}(T) \\ \boldsymbol{x}^{(2)}(1) & \dots & \boldsymbol{x}^{(2)}(T) \\ \vdots & & \vdots \\ \boldsymbol{x}^{(N_r)}(1) & \dots & \boldsymbol{x}^{(N_r)}(T) \end{bmatrix} = \boldsymbol{X} \in \mathbb{R}^{N_r N_n \times T} \tag{14}$$

Using this state matrix, the training takes place as in Equation 9 where, in this case, the identity matrix $\boldsymbol{I} \in \mathbb{R}^{N_r N_n \times N_r N_n}$.

## 3.2 Review of Anomaly detection

Anomaly detection, or outlier detection, is an important problem that refers to the task of identifying patterns in data that do not comply with the expected behaviour [11, 12, 13, 14]. Many industrial real problems focus on finding anomalies from time series, that is data depending on time [13, 15, 16, 17, 18, 19].

There are three distinct categories of anomaly approaches, depending on the existence of labels in training data, that is supervised, semi supervised or unsupervised anomaly detection. This paper deals with the last category of unsupervised anomaly detection, which is the most difficult scenario due to the absence of any label of the data.

### 3.2.1 Anomaly detection using Echo State Networks

Echo State Networks have been used for unsupervised anomaly detection, in particular for situations where the network is trained only with data belonging to the normal class (for example in situations where the available data is imbalanced in the numbers for the normal and anomalous classes) [20, 21, 22, 23, 24]

## 3.3 Review of Echo State Networks online learning

Echo State Network have been used for online training applications. The reason for this is the low computational complexity required for the ESN training, which does not require the more computationally complex and time requiring process of stochastic gradient descent, used in the most cases of deep learning usage [4]. Examples of online trained ESN applications are found in in [25, 26, 27, 28, 29].

## 3.4 Review of oil leaks detection, thresholding and block-based processing

Oil leaks detection has been used in the context of satellites synthetic aperture radar images of the sea surface. One important process in these studies, that happens before the evaluation of the presence or absence of leaks, is the feature extraction, that can take place through thresholding [30, 31].

Another common process for these studies, also performed before the anomaly detection, is the image segmentation [30, 32, 33, 34], that divides the images blocks (of fixed or variable dimensions) into sub-images. The segmentation also permits to discard not useful blocks, like land zones [34].

In these cases, however, blocks have been used as a single images, uncorrelated to the other blocks. Moreover, the networks used in these studies were not recurrent, since the images were not analyzed in a temporal fashion.

Apart from these fields of study, block based-image processing has been used in Compressed Sensing [35], Noise Removal [36], Image Recapture [37] and Artifact Removal and Image Compression [38, 39].

### 3.5 Review of Network quantization

Quantization is the process of approximating a continuous signal by a set of discrete symbols or integer values. Most networks are trained using signle precision floating point representation (*fp32*). In our case an *fp32* or even *fp64* representation has greater precision than needed, requiring 4 to 8 times more memory than an integer 8 bits models or 32 to 64 times than a integer 1 bit models. Therefore, converting *fp32* parameters to less precise bit representations, such as *int8* and binary (*int1*), can significantly reduce memory footprint, power consumption and offer the opportunity of an increased execution speed. However, this cannot be achieved by compromising the accuracy of the *fp32* of *fp64* precision models.

Quantization is categorized into two areas: 1) quantization aware training (QAT) and 2) post training quantization (PTQ). The difference depends on whether quantization is affecting weights learning during training or not. Alternatively, it is also possible to categorize quantization by where data are grouped for quantization: 1) layer-wise and 2) channelwise. A further classifiation based on bit-depth, such N-bit quantization. With respect to network quantization, examples can be found in [40, 41]. [9] presents a study of an ESN designed with states represented by integer numbers and a binary recurrence matrix.

### 3.6 Review of Extreme Learning Machines

Extreme Learning Machines (briefly ELM) share similarities with ESN and DeepESN. They are defined as feedforward, not recurrent neural networks for classification, regression, clustering, sparse approximation, compression and feature learning with a single layer or multiple layers of hidden nodes, in which the parameters of the hidden nodes (not just the weights connecting inputs to hidden nodes) do not need to be tuned. These hidden nodes can be randomly assigned and kept fixed (i.e. they are random projection but with nonlinear transforms), just like it can happen with ESN and DeepESN. In most cases, the output weights of hidden nodes are learned in a single step, which essentially amounts to learning a linear model. For this reason, ELM permit inference and learning need to be performed in a fixed and

predictable amount of time and memory. The name "Extreme Learning Machines" was given to such models by its main inventors Guang-Bin Huang, Qin-Yu Zhu and Chee-Kheong Siew. According to their creators, these models are able to produce good generalization performance and learn thousands of times faster than networks trained using backpropagation [42]. In literature, it is also shown that these models can outperform support vector machines in both classification and regression applications [43, 44, 45].

The output function of an ELM is

$$f_L(x) = \sum_{i=1}^{L} \beta_i h_i(a_i, b_i, x) = h(x)\beta \tag{15}$$

where $\beta = [\beta_1, \ldots, \beta_L]^T$ is the output weight vector and
$h(x) = [h_1(a_1, b_1, x), \ldots, h_L(a_L, b_L, x)$ is the output vector of the hidden layer, with hidden node parameters $(a_i, b_i)$. Basically, an ELM is trained in two main stages: (1) random weights assignment to the hidden layers nodes, and (2) linear parameters solving. In the first stage the hidden node parameters are randomly generated, independently from the training data, according to any continuous probability distribution instead of being explicitly trained. In the second stage of ELM learning, the weights $\beta$ connecting to the hidden layer and the output layer and derived by solving the linear equation

$$T = H\beta \tag{16}$$

where $H$ is the hidden layer output matrix and $T$ is the training data target matrix [43].

As a comparison with the network designed for this study, Subsection 9.4.2 reports the results obtained using the same proposed BBS-ESN architecture, but without the reservoirs recurrent connections.

# 4 Dataset creation

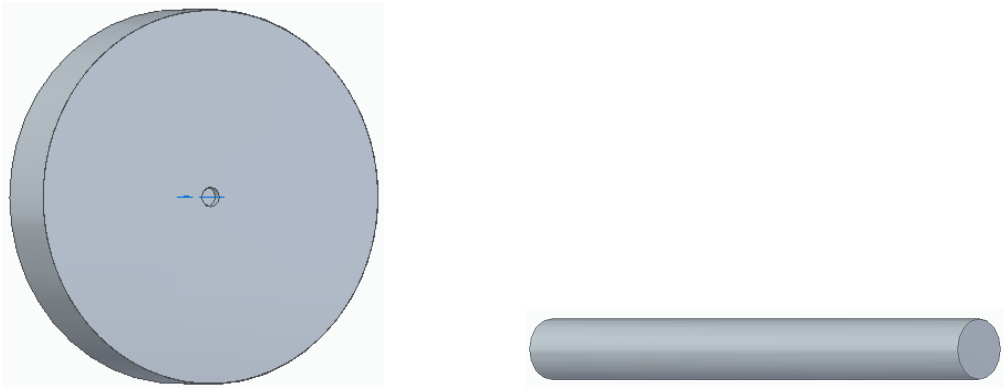
A public dataset comprehensive of various framings of the junction of interest in this study, in presence and absence of oil leaks, was not available. This is why it has been created a specific one, consisting of a collection of synthetic images simulating photographies that would be acquired by a live image sensors, representing normality (absence of leaks) and abnormality (presence of leaks). The images have been produced through graphic 3D rendering in RGB format. The images have been produced in various framings, viewpoints and metallic surfaces textures and, for the anomalous images, various oil leak colors. Subsequent preprocessing has been performed using Matlab, also producing grayscale and binary images generation, using the RGB as starting point. Finally, data augmentation has been performed in Python, for any further data quantity and quality enhancing.

The dataset is publicly available at

https://data.mendeley.com/datasets/nbxzxn3ffk/1.

## 4.1 3D model

The starting point for the renderings has been two 3D parts generated in SolidEdge 2020 [1], shown in Figure 6. The bracket part in Figure 6a is a cylinder with an internal cylindrical cavity. The cylinder has dimensions of 2,000mm of diameter and 400mm of height and the cavity hole has a diameter of 105mm and a depth of 50mm.
The shaft part in Figure 6b is also cylindrical, with a diameter of 100mm and a height of 1,000mm.



(a) Bracket part used for the rendering assembly. It represents the external bracket of the power generator

(b) Shaft part used for the rendering assembly. It represent the high speed shaft

Figure 6: Parts used to compose the assembly used as rendering basis

The two parts compose an assembly, shown in Figure 7. The shaft part is

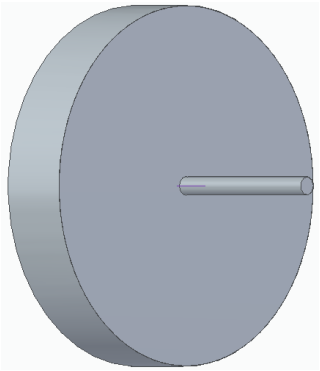


Figure 7: Assembly composed with the parts in Figure 6

positioned inside the cavity of the bracket part. The diameters of the shaft part and of the cavity in the bracket part are not the same: they are respectively 100mm and

[1] https://solidedge.siemens.com/en/

105mm. This has been done in order to simulate some space in their coupling, as can be seen in the example figures of Section 4.3.

## 4.2 Rendering

### 4.2.1 Leaks images

Leaks images have been produced using the image manipulation software Gimp[2]. These images have been produced in *.png format because of the importance to generate them with a transparent background. Brush tools of white color have been used to produce the leak images. One or more leaks have been produced for each `Group` folder for images with leak. Only in some cases leak images were generated to be positioned, also, on the the bracket part of Figure 6b and not only on the shaft part of Figure 6a. In Figure 8 are represented examples of leak images, with the leaks colored in black for a visibility matter.

Once generated the leak images, they were imported in KeyShot9 as textures. They have been changed in color, positioned, resized and wrapped (for the leaks to be used in the shaft part). The leaks textures have also been edited adding with bump mapping and displacement mapping to let them to appear three-dimensional in the renderings.

### 4.2.2 Light, background and virtual cameras specifics

The environment has been chosen as a completely white background, spreading uniform and diffurse white light, while subsequent regulations on the brightness level made the background of a light gray color, giving the images a more natural look.

The 3D model has been imported into Keyshot9 and the renderings have been performed using 16 virtual cameras, all with the same aperture of 39.6°, that have been kept in fixed positions for the generation of all the images.

[2]https://www.gimp.org/

(a) Leak image used for the shaft part in `Group_12`

(b) Leak image used for the bracket part in `Group_12`

(c) Leak image used for the shaft part in `Group_16`

Figure 8: Leaks images examples. All the leak images have been produced as white *.png images with transparent background. Here they are represented in black for sake of visibility.

Because of the symmetrical nature of the problem it has been decided not to render images from the right side of the junction.

More than one framing has been used in order to simulate different camera positions. In fact, during the system setup the cameras, while remaining in fixed positions after their positioning, may be placed with a variable framing of the junction.

### 4.2.3 Rendering specifics

The rendering has been performed using a CPU, specifically an Intel i5-6200U, with two cores and a clock frequency 2.3 GHz. All the renderings have been performed with the following specifics:

**Samples:** images have been rendered with 4 samples (i.e. calculated 4 times in order to increase accuracy)[46]

**Ray bounces:** the number of ray bounces, that is the time light reflexes are calculated, have been set to 6. (that is the standard number).

**Pixel blur:** the amount of blur added to avoid over-sharp images. It has been disabled in order to keep sharp shapes.

**Anti-aliasing:** used to smooth jagged edges. It has been set to 1, as indicated as sufficient in [46].

**Shadow quality:** it has been kept to the minimum, that is 1, in order to reduce the rendering times, since in [46] is reported the dramatic time increase due to shadow quality raising.

**Resolution:** all the images obtained with rendering have the same resolution of $1024 \times 768$ pixels.

## 4.3 Images disposition in the Dataset and examples

The dataset is contained in the main folder `Oil_leak_dataset`. Inside this folder is contained the sub-foder `Junction_images`, where are located 22 directories of images, named `Group_1` to `Group_22`, and the Matlab preprocessing script, named `Preprocessing.m`. Groups from 1 to 20 are dedicated to images with leaks, each one containing images with a different leak shape, while groups 21 and 22 contain images without leak.

The main folder `Oil_leak_dataset` also contains two Google Colab notebooks for the augmentation of the grayscale images (`Oil_leak_grayscale_augmentation.ipynb`) and of the binary images (`Oil_leak_binary_augmentation.ipynb`). Both the grayscale and the binary images are obtained using the preprocessing script.
The directoty tree is reported below:

```
Oil_leak_dataset
└── Junction_images
    └── Group_1
```

```
│
│   ├── ...
│   ├── Group_22
│   └── Preprocessing.m
├── Oil_leak_grayscale_augmentation.ipynb
└── Oil_leak_binary_augmentation.ipynb
```

### 4.3.1 Images with oil leak

Groups from 1 to 20 are dedicated to images with leaks. The disposition of the images with oil leak inside the directory `Junction_images`, before the application of the preprocessing, is:

```
Junction_images
├── Group_1
├── ...
└── Group_20
    ├── set_1
    ├── ...
    └── set_6
        └── natural_light
            ├── g20s6.1nat.jpg
            ├── ...
            └── g20s6.16nat.jpg
```

The images in each folder are named as follows:
`g<group number>s<set number>.<image number><label>.<extension>`.

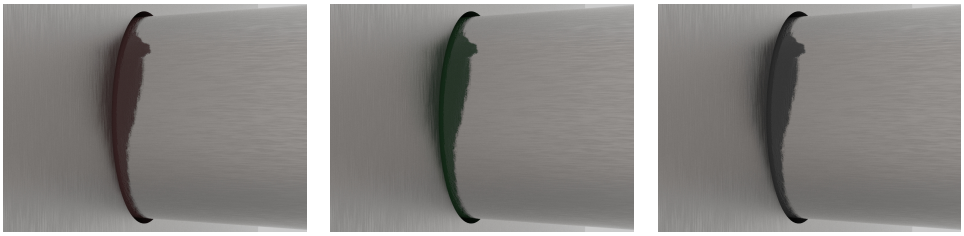
In this notation, `g` stays for "Group" and `s` stays for "set". After the point follow the image number and a label. The labels are univocally associated with the type of preprocessing or to the images obtained directly from rendering, as in this case. In these images, before applying the preprocessing, the label is `nat`, brief for "natural light".

In each of the groups the leak shape is the same. It is important to point out that, in the entire dataset, images associated with the same `<image number>` are generated with the same framing.

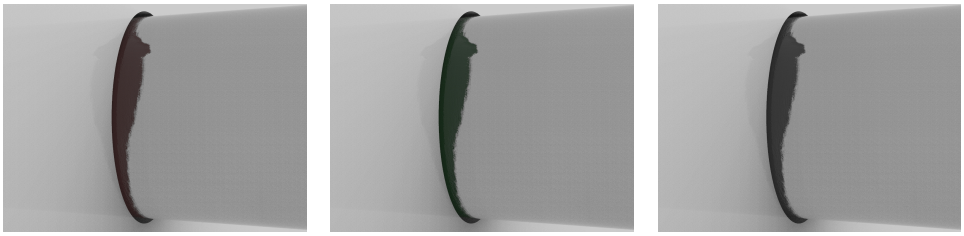
Each set, from `set_1` to `set_6`, contains 16 RGB images, specifically:

- `set_1`: not painted steel texture, brown leak
- `set_2`: not painted steel texture, green leak
- `set_3`: not painted steel texture, gray leak
- `set_4`: white painted steel texture, brown leak
- `set_5`: white painted steel texture, green leak
- `set_6`: white painted steel texture, gray leak

Figure 9 shows examples of images with the same framing (i.e. with the same `<image number>`) of `Group_16`, from all the 6 sets (`g16s<set number 1-6>.2nat.jpg`).



(a) not painted steel texture, brown leak (b) not painted steel texture, green leak (c) not painted steel texture, gray leak



(d) white painted steel texture, brown leak (e) white painted steel texture, green leak (f) white painted steel texture, gray leak

Figure 9: Natural light (i.e. not preprocessed) images from `Group_16`, one for each set. Each image has the same `<image number>` in the sets and as a consequence they have the same framing.

### 4.3.2 Images without oil leak

Groups 21 and 22 contain images without oil leaks. `Group_21` contains images rendered with not painted metallic texture, while `Group_22` contains images rendered with white paint metallic texture.

The directory tree of the images not presenting oil leak, before applying preprocessing, is the following:

```
Junction_images
├── Group_21
│   └── set_1
│       └── natural_light
│           ├── g21s1.1nat.jpg
│           ├── ...
│           └── g21s1.16nat.jpg
└── Group_22
    └── set_1
        └── natural_light
            ├── g22s1.1nat.jpg
            ├── ...
            └── g22s1.16nat.jpg
```

, with only one set per group (in both cases named `set_1`). Figure 10 shows examples of images not presenting oil leaks, with not painted steel texture.

(a) white painted steel texture, no leak (b) not painted steel texture, no leak

Figure 10: Natural light (i.e. not preprocessed) images without leak

### 4.4 Matlab preprocessing

The Matlab preprocessing script can been applied to the rendered images. The script has been designed with 2020b Matlab version, using the Image Processing Toolbox [47].

The folder structure of the images with oil leaks, after the appliacation of the preprocessing, is:

```
Junction_images
├── Group_1
├── ...
└── Group_20
    ├── binary_1
    │   ├── g20s1.1bin.png
    │   └── ...
    ├── binary_2
    │   ├── g20s4.1bin.png
    │   └── ...
    ├── grayscale_1
    │   ├── g20s1.1gs.jpg
    │   └── ...
    ├── grayscale_2
    │   ├── g20s4.1gs.jpg
    │   └── ...
    ├── set_1
    ├── ...
    └── set_6
        ├── cold_light
        │   ├── g20s6.1cold.jpg
        │   └── ...
        ├── low_light_1
        │   ├── g20s6.1low1.jpg
        │   └── ...
        └── low_light_2
```

```
│   ├── g20s6.1low2.jpg
│   └── ...
├── natural_light
│   ├── g20s6.1nat.jpg
│   └── ...
├── oversaturated
│   ├── g20s6.1sat.jpg
│   └── ...
└── warm_light
    ├── g20s6.1warm.jpg
    └── ...
```

As for the images without leak, after the preprocessing the direcrory tree is the one below:

```
Junction_images
├── Group_21
│   ├── binary_1
│   │   ├── g21s1.1bin.png
│   │   └── ...
│   ├── grayscale_1
│   │   ├── g21s1.1bin.png
│   │   └── ...
│   └── set_1
│       ├── cold_light
│       │   ├── g21s1.1cold.jpg
│       │   └── ...
│       ├── low_light_1
│       ├── low_light_2
│       ├── natural_light
│       ├── oversaturated
│       └── warm_light
└── Group_22
    ├── binary_1
    │   ├── g22s1.1bin.png
    │   └── ...
    ├── grayscale_1
    │   ├── g22s1.1bin.png
    │   └── ...
    └── set_1
        ├── cold_light
        │   ├── g22s1.1cold.jpg
        │   └── ...
```

```
├── low_light_1
├── low_light_2
├── natural_light
├── oversaturated
└── warm_light
```

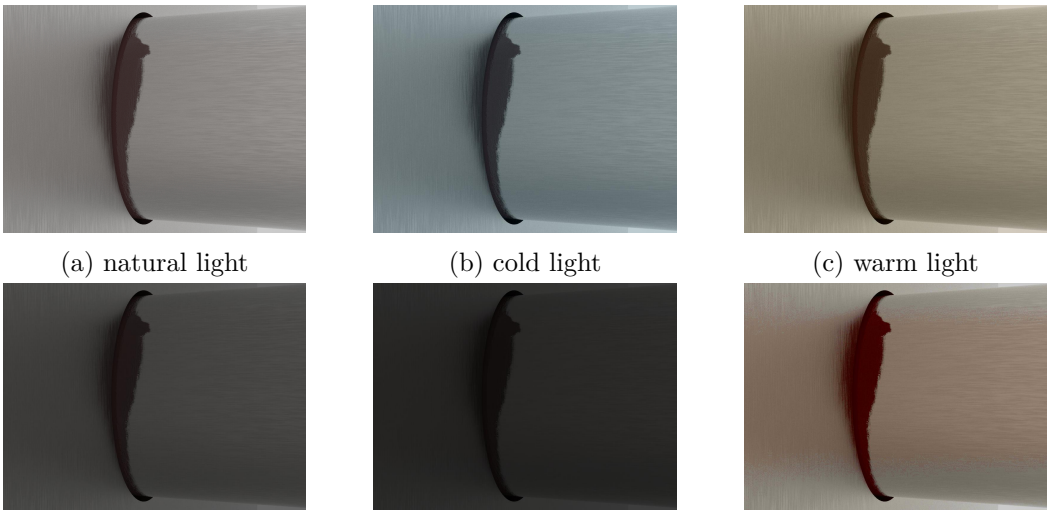
### 4.4.1 RGB images from preprocessing

The images generated from the RGB preprocessing are in *.jpg format.

To the images of the `natural_light` folders, inside each set, are applied 5 different RGB preprocessings. To each processing a `<label>` is associated in the image name: `g<group number>s<set number>.<image number><label>.<extension>`, substituting the label `nat`. These preprocessings are:

**luminance value decrease:** 0.5 times the original. Associated label: `low1`.
**luminance value decrease:** 0.3 times the original. Associated label: `low2`.
**saturation enhancement** of 5 times the original. Associated label: `sat`.
**warmer light temperature:** +5 points in red, -15 points in blue. Associated label: `cold`.
**colder light temperature:** -15 points in red, +5 points in blue. Associated label `warm`.

Figure 11 shows examples of the same RGB image (`g16s1.2nat.jpg`), not painted steel texture and brown leak, to which are applied the 5 RGB preprocessings.



(a) natural light (b) cold light (c) warm light

(d) low light 1 (e) low light 2 (f) enhanced saturation

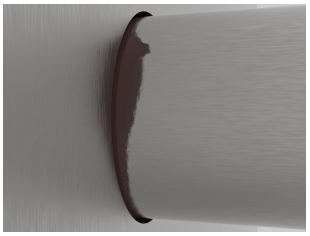
Figure 11: RGB Images from `Group_16`, one for each set. In each set it is shown an image with the same `image number` (and as a consequence the same framing).

### 4.4.2 Grayscale images from preprocessing

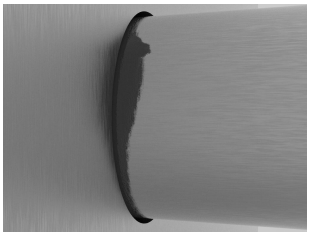
The images obtained frmo the grayscale conversion are in *.jpg format. The associated label is `gs`. The conversion from RGB format to grayscale eliminates hue and saturation, retaining the luminance value of the image [48].

The grayscale images in the directories `grayscale_1` are obtained from the `set_1` images of each group, while the images in `grayscale_2` are obtained from the `set_4` images of the same group. This is why the `<set_number>` of the images in the `grayscale_2` folders is 4.

For `Group_21` and `Group_22`, that contain images without leak, there is only one set of images (Subsection 4.3.2). For this reason only one directory of grayscale images will be created, named `grayscale_1`, as shown in the directory tree for images without oil leaks in Subsection 4.4. Figure 12 shows an example of grayscale conversion (and subsequent binarization, that will be explained in Subsection 4.4.3).

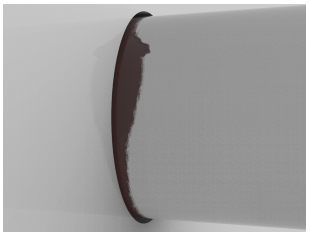


(a) Not painted steel texture, brown leak, not preprocessed

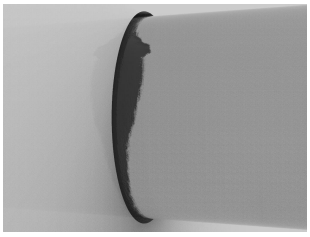


(b) Grayscale preprocessing of the image at the left

(c) Binary version of the image at the left



(d) White painted steel texture, brown leak, not preprocessed



(e) Grayscale version of the image at the left

(f) Binary version of the image at the left

Figure 12: Images from `Group_16`: `set_1` and `set_4` (a and d) converted to grayscale (b and e). The grayscale images are binarized in c and f.

### 4.4.3 Binary images from preprocessing

The binary images obtained from the preprocessing are in *.png format and associated label is `bin`. The binary images have been saved in *.png format because *.jpg format does not allow to have a distribution of only 0 and 1 values. The binary images are obtained from the grayscale ones and the conversion is performed calculating a threshold using the Otsu's method [49].

The images in the directories `binary_1` are obtained from the `grayscale_1` direc-

tories from each group, while the images in `binary_2` are obtained from the directory `grayscale_2` of the same group. Since the images from the directories `grayscale_2` are obtained from `set_4` of the respective group, the images `<set_number>` is 4.

As explained in Subsection 4.4.2 `Group_21` and `Group_22` will contain, after preprocessing, only one grayscale images folder, named `grayscale_1`. For this reason `Group_21` and `Group_22`, after preprocessing, will contain only one binary images folder, named `binary_1`. Figure 12 shows an example of the process of grayscale conversion and binarization.

## 4.5 Python data augmentation

Data Augmentation is a process that uses techniques to enhance the data quantity and quality, producing useful variability in order more accurate deep neural networks [50]. Two scripts for augmentation are provided: one for grayscale images and one for binary images, since these two images classes are in lower numbers after preprocessing. The data augmentation scripts have been designed with Python 3.7.11 and use the ImageDataGenerator class, provided by the Keras (version 2.5.0) library [51]. The augmentation functions used are methods of the ImageDataGenerator class and other functions relying on Tensorflow (version 2.5.0) and scikit-image (version 0.16.2).

Both the augmentation scripts perform the following operations:

- Download the relative images directory (binary or grayscale)
- Import useful libraries like Numpy, Tensorflow, Scikit-Image, os, Pathlib
- Enable/disable the augmentation functions commenting/uncommenting from transformation names lists
- Definition of the augmentation that are not methods of the ImageDataGenerator class (using Tensorflow and scikit-image)
- Perform the augmentation functions
- Visualize the augmented images, compared to the original ones (this step is optional).
- Save the augmented data. The dataset can be saved to the user's Google Drive.

The transformations that are performed are reported in Table 1
The default specifications of these transformations are: erosion with disk filter of radius 5, rotation range angle of 40° (with nearest fill mode), width shift range of 0.2 (with nearest fill mode), height shift range of 0.2 (with nearest fill mode), zoom range of 0.2, shear range of 0.2, salt&pepper noise at 1% and 5% percentages, gaussian noise with mean 0 and variance 0.01. The default values can be changed by the user.

The augmentation process for grayscale images has been designed with the aim of applying noise addition to both the original images and the spatially augmented ones (scheme in Figure 13).

Table 1: Table of transformations applied to binary and grayscale images and the corresponding labels added to the augmented images.

| | ***Grayscale*** | ***Binary*** | ***Corresponding label*** |
|---|---|---|---|
| Erosion filter | x | ✓ | `erosion` |
| Rotation | ✓ | ✓ | `rotation` |
| Horizontal flip | ✓ | ✓ | `horizontalFlip` |
| Vertical flip | ✓ | ✓ | `verticalFlip` |
| Height shift | ✓ | ✓ | `heightShift` |
| Width shift | ✓ | ✓ | `widthShift` |
| Zoom | ✓ | ✓ | `zoom` |
| Shear range | ✓ | ✓ | `shearRange` |
| Salt and pepper noise 1% | ✓ | ✓ | `s&p1%` |
| Salt and pepper noise 5% | ✓ | ✓ | `s&p5%` |
| Gaussian noise | ✓ | x | `gauss` |

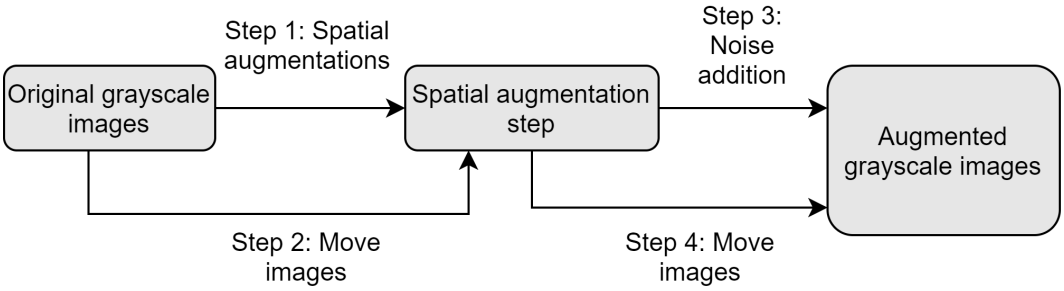



Figure 13: Diagrams of the augmentations pipeline for grayscale images. In this way the noise addidion is performed to both the spatially augmented images and the original ones.

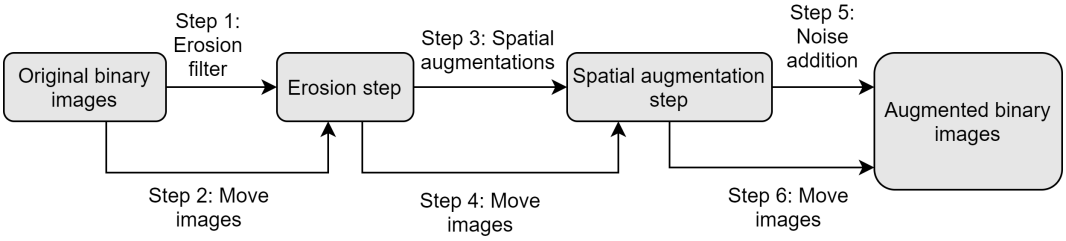



Figure 14: Diagrams of the augmentations pipeline for binary images. The spatial augmentations are performed after the erosion filter application to both the eroded and not eroded images. All these images are subsequently processed for saòt&pepper noise addidion.

As for the binary images, the augmentation has been designed with the aim of to applying the erosion filter to the original images, subsequently apply the spatial

augmentations to the eroded and original images and finally apply the noise addition to all these images combinations. The scheme in Figure 14 synthesizes the pipeline. For both the scripts the original directory structure is preserved.

When an augmentation is applied a label(indicated in Table 1) is added to the image name, preceded by and underscore.

### 4.6 Images quantity

The number of images obtained through rendering, after preprocessing and after data augmentation are reported, respectively, in Table 2, Table 3 and Table 4.

Table 2: Number of images obtained through rendering

| | ***RGB*** | ***Grayscale*** | ***Binary*** |
|---|---|---|---|
| With leak | 1,920 | - | - |
| Without leak | 32 | - | - |

Table 3: Number of images after preprocessing

| | ***RGB*** | ***Grayscale*** | ***Binary*** |
|---|---|---|---|
| With leak | 11,520 | 640 | 640 |
| Without leak | 192 | 32 | 32 |

Table 4: Number of images after data augmentation

| | ***RGB*** | ***Grayscale*** | ***Binary*** |
|---|---|---|---|
| With leak | 11,520 | 20,480 | 30,720 |
| Without leak | 192 | 1,024 | 1,536 |

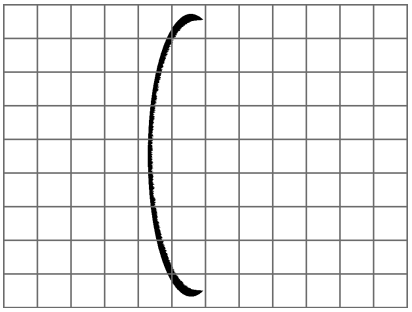
## 5 Images usage

To make an estimation of the camera distances in order to have a 360° view, it has been took advantage of the cylindrical nature of the junction. As a reminder, the virtual cameras used have a field of view of 39.6°, as explained in subsubsection 4.2.3. In the hypothesis of positioning the cameras with view centered perpendicularly to the shaft surface, the condition for a with camera with that field of view to have a 90° view of the shaft is to be at a distance from the shaft radius of $l = (r\sqrt{2}/2) \cdot \left(1 + \tan^{-1}(39.6/2)\right) = 2.67\,r$ (with $r$ the shaft radius).

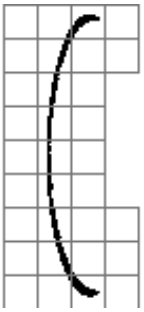
For the images in this dataset, though, the distances of the virtual cameras are variable and permit to cover a view of 180° of the shaft, generating as a consequence images that overlap in the covered zones.

For the network design, using binary pictures as inputs, the images original size has been scaled reducing each side to approximately 20% of the original size, to $192 \times 144$ resolution. Resizing has been used both for the baseline and the quantized

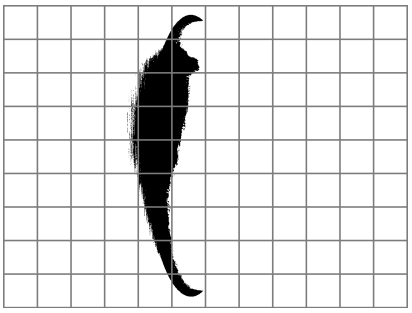
versions of the BBS-ESN and it has been performed using the bilinear interpolation technique [52]. From the assumption of the cameras to be mounted in fixed positions, as shown in Figure 2a, the acquired images can be decomposed through a fixed mask in non overlapping blocks and can be read from the sensor only those blocks which enclose portions of interest of the junction. Figure 15 shows examples of normal and anomalous binary images in the same framing, resized and divided in blocks (both RGB and binarized). Only the blocks outlining the junction have been chosen to be used as network inputs for both learning and detection of anomalies. In fact, they are the only ones carrying useful information for the purpose of oil leaks detection.



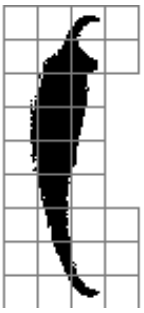
(a) Binary normal image divided in blocks, $1024 \times 768$ pixels



(b) Binary normal image with only blocks outlining the junction, $192 \times 144$ pixels



(c) Binary anomalous image divided in blocks, $1024 \times 768$ pixels



(d) Binary anomalous image with only blocks outlining the junction, $192 \times 144$ pixels

Figure 15: Binary normal and anomalous images are resized, with selected subset of blocks subject of further processing.

In order to test the BBS-ESN networks simulating the temporal propagation of oil leaks anomalies with a gradual transition, image morphing has been used [53, 54]. Normal images have been used as starting frame and anomalous ones with the same framing (and original metallic texture in the RGB images) have been used as ending frame. The kind of morphing used is called "Differentiable morphing", performing morphing without reference points, using a neural network to calculate warp maps [55].

10 sequences have been produced, each one constituted by 100 images. More than one framing has been used in order to simulate different camera positions. In

fact, during the system setup the cameras, while remaining in fixed positions, may be placed with a variable framing of the junction. The mean blocks number for the sequences used is 29.4. The images used at the starting and the ending of the sequences are reported in Table 5.

Table 5: Images used as starting frame and ending frame of the morphing sequences

| ***Morphing sequence*** | ***Starting image*** | ***Ending image*** |
|---|---|---|
| 1 | g21s1.1bin | g4s1.1bin |
| 2 | g21s1.2bin | g2s1.2bin |
| 3 | g22s1.3bin | g2s4.3bin |
| 4 | g21s1.4bin | g4s1.4bin |
| 5 | g21s1.5bin | g5s1.5bin |
| 6 | g21s1.6bin | g6s1.6bin |
| 7 | g22s1.7bin | g7s4.7bin |
| 8 | g22s1.8bin | g8s4.8bin |
| 9 | g21s1.9bin | g9s1.9bin |
| 10 | g21s1.10bin | g10s1.10bin |

# 6 Proposed Block based Binary Shallow Echo State Network (BBS-ESN)

Two network versions have been designed, both with the same architecture but presenting differences in the weights and the non-linearity precisions. They present, also, differences in the reservoirs matrices connectivity (i.e. percetage of non-zero weights). The starting point in the network design has been the baseline BBS-ESN, from which the quantized BBS-ESN has been derived.

## 6.1 BBS-ESN training

The network training modality, for both the proposed network, is based on the assumption that the image sensor is initially set and turned on framing the junction while it is in normal state (i.e. without leaks). The rationale is that the network, through training, aims at approximating the input image in a situation without anomalies, while the framing of the camera remains the fixed. Once the learning is concluded, the network is put in inference mode.

An anomaly condition will be asynchronously identified in the condition of the output being significantly different from the input. This condition would occur because the image under analysis is significantly different from the ones used by the training step, therefore being anomalous. This approach is similar to the ones adopted in [20, 21, 22] using ESN for anomaly detection.

Due to this formulation of network training, the network states caused by the

input blocks are stacked into the state matrix $\boldsymbol{X}$, while the input blocks are used as train targets $\boldsymbol{y}(t)$ (Equation 2). Therefore, the states are disposed as in Equation 17:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}^{(1)}(1) & \dots & \boldsymbol{x}^{(1)}(B) \\ \boldsymbol{x}^{(2)}(1) & \dots & \boldsymbol{x}^{(2)}(B) \end{bmatrix}, \tag{17}$$

begin $B$ the number of blocks used for the training. The matrix of the train targets $\boldsymbol{Y}_{targets}$ is represented in Equation 18

$$\boldsymbol{Y}_{target} = \begin{bmatrix} \boldsymbol{u}(1) & \dots & \boldsymbol{u}(B) \end{bmatrix}, \tag{18}$$

where the blocks $\boldsymbol{u}(t)$ are acquired from more than one image and are placed in the same order they have been used as network inputs. These two matrices are used to compute $\boldsymbol{W}_{out}$ as in Equation 9. The blocks order of Equation 18 can vary and the possibilities are explained in section 8. The choice on the number of blocks fell on 80, which reason is explained in subsubsection 9.2.1. As for the choice of stacking the states vertically, experiments have been conducted using the visualization technique t-SNE [56] on the network states, treating them as 256-dimensional values and embedding them in a 2 dimensional map. These experiments shown a separation of the mapping zones of the 2 reservoir states, justifying the assignation of different weights to them (as the scheme in Figure 16 proposes).

## 6.2 Architecture overview

The BBS-ESN was designed, in both full precision baseline and a quantized version, with two reservoir layers. Figure 16 shows the top level view of the BBS-ESN.

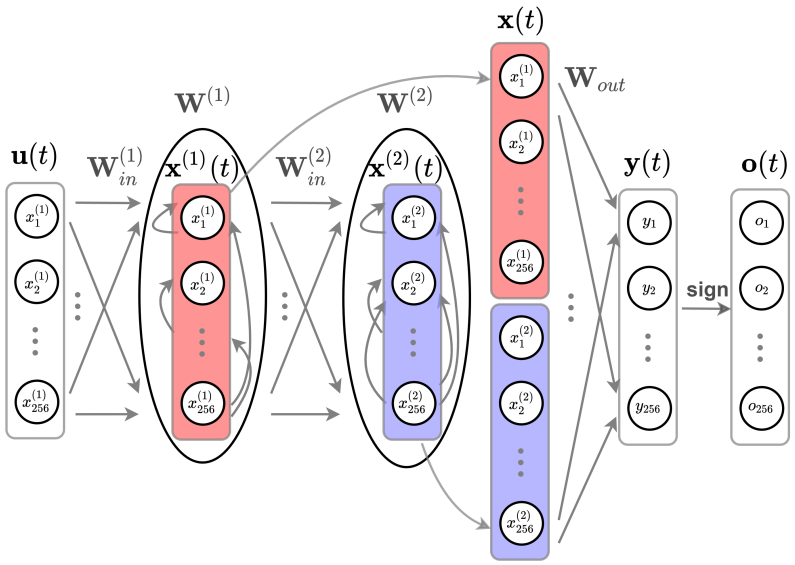



Figure 16: Top architecture of the BBS-ESN, that uses 2 reservoir layers.

Both the BBS-ESN versions use two reservoirs of 256 neurons each. The inputs

are binary column vectors of dimension $256 \times 1$ (obtained from images blocks of dimension $16 \times 16$ that are flattened). The BBS-ESN output is also a binary column vector of dimension $256 \times 1$.

The network pipeline follow the description below:

- **First reservoir state:** The input to the network, $\boldsymbol{u}(t)$, is a binary image block (of dimension $16 \times 16$) flattened in a column vector of dimension $256 \times 1$. *The binary numbers are represented by -1 and 1 values.* The input is mapped into the first reservoir layer by the matrix $\boldsymbol{W}_{in}^{(1)}$ and the result is added to the reservoir state at the previous time $\boldsymbol{x}^{(1)}(t-1)$, multiplied by a reservoir matrix $\boldsymbol{W}^{(1)}$. The weights of $\boldsymbol{W}^{(1)}$ represent the weighted connection inside each reservoir, from which the state of each neuron depends on the state of the others. A non-linearity is applied to the result of the sum (Equation 11).

- **Second reservoir state:** The same operations are performed in the second reservoir layer, except for its input that is the first reservoir state at the same temporal instant (Equation 11).

- **Network readout:** The network readout $\boldsymbol{y}(t)$ is computed using both the reservoir states vertically stacked in a column vector, as per Equation 19,

$$\boldsymbol{x}(t) = \begin{bmatrix} \boldsymbol{x}^{(1)}(t) & \boldsymbol{x}^{(2)}(t) \end{bmatrix}. \tag{19}$$

  The BBS-ESN output is expressed by Equation 20

$$\boldsymbol{y}(t) = \boldsymbol{W}_{out}\boldsymbol{x}(t). \tag{20}$$

  The matrix $\boldsymbol{W}_{out}$ is the only element of the network that is subject to training.

- **Network output with sign extraction:** Finally, the sign extraction is applied to the network readout $\boldsymbol{y}(t)$ (Equation 21):

$$\boldsymbol{o}(t) = sign(\boldsymbol{y}(t)). \tag{21}$$

The weights of the matrices $\boldsymbol{W}_{in}^{(1)}$, $\boldsymbol{W}^{(1)}$, $\boldsymbol{W}_{in}^{(2)}$ and $\boldsymbol{W}^{(2)}$ are composed by randomly instantiated weights that are kept always fixed during learning and inference. The sign extraction of Equation 21 does not play a role in the training.

### 6.3 Baseline BBS-ESN implementation

The baseline BBS-ESN relies on double precision floating point weights and computations. All the matrices are fully connected, that means that they have all non-zero coefficients. The matrices $\boldsymbol{W}_{in}^{(1)}$, $\boldsymbol{W}^{(1)}$, $\boldsymbol{W}_{in}^{(2)}$ and $\boldsymbol{W}^{(2)}$ are instantiated (and then kept fixed) with a random uniform distribution between -1 and 1, using the MT19937 pseudorandom number generator [57]. The coefficients of $\boldsymbol{W}^{(1)}$ and

$\boldsymbol{W}^{(2)}$ are subsequently rescaled in order for the matrices to have spectral radius [8, 5] $\rho(\boldsymbol{W}^{(1)}) = \rho(\boldsymbol{W}^{(2)}) = 0.95$.

The baseline BBS-ESN parameters are summarized in Table 6, Table 7 and Table 8 ("bin[-1,1]", from here on, stays for "binary distribution of -1 and 1"). In Table 7 is reported that the weights matrices $\boldsymbol{W}_{in}^{(1)}$, $\boldsymbol{W}_{in}^{(2)}$, $\boldsymbol{W}^{(1)}$, and $\boldsymbol{W}^{(2)}$ are to be saved in Flash memory. The reason is that, since they are random instantiated values that will never change, they can be used as constant values.

The readout training of the baseline BBS-ESN implements the operations reported in Table 9. These operations are the ones performed in Equation 9, using $\alpha^2 = 1$. In the table the $\boldsymbol{K}$ matrix inversion computations are not reported, since it can be implemented with many methods. It is reported the order of magnitude of the operations, in relation to the matrix dimension.

## 6.4 Quantized BBS-ESN implementation

### 6.4.1 Binarization of the reservoirs

As reported in [10], the connectivity of the reservoir $\boldsymbol{W}$, in an ESN, can be reduced leaving connections (i.e. non-zero weights) only in the diagonal matirx. In [9] the reservoir weights have been set to identity matrices with all the rows shifted of a fixed position. Instead, in this study both the reservoir matrices, $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$, have been set as an identity matrices. In this case $\rho(\boldsymbol{W}^{(1)}) = 1$ and $\rho(\boldsymbol{W}^{(2)}) = 1$. Therefore, the condition on the spectral radius has been relaxed from $\rho(\boldsymbol{W}^{(1)}) < 1$ and $\rho(\boldsymbol{W}^{(2)}) < 1$, as in [8, 5], to $\rho(\boldsymbol{W}^{(1)}) \leq 1$ and $\rho(\boldsymbol{W}^{(2)}) \leq 1$, as in [9],

This weights distributions on $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$ mean that the state of a neuron depends only on the state of the same neuron at the preceding instant.

Furthermore, reservoir matrices as identity matrices have the advantage that their multiplications do not need to be executed, nor it is required memory space to save its values. For this reason these matrices are not listed in Table 10 and Table 11.

### 6.4.2 Binarization of the input matrices

$\boldsymbol{W}_{in}^{(1)}$ and $\boldsymbol{W}_{in}^{(2)}$ have been instantiated with uniform distributions of binary values (-1 and 1). The distributions have been obtained using the same pseudorandom number generator used for the baseline BBS-ESN, that is the MT11937 [57].

### 6.4.3 Quantized readout layer

The readout weights have been quantized to *int8*, with a procedure that follows the principle of normalization. All the elements of the matrix are divided for the smaller of the matrix entries, taken in modulus, called from now $\min(|W_{out\ i,j}|)$.

It has been assessed, empirically, that $\min(|W_{out\ i,j}|)$ is always smaller than 1.

Table 6: Baseline BBS-ESN parameters

| | |
|---|---|
| ***Blocks size*** | 16 × 16 pixels |
| ***Blocks values distribution*** | bin.[-1, 1] |
| ***Input vector units*** | 256 × 1 pixels* |
| ***Number of reservoirs*** | 2 |
| ***Neurons per reservoirs*** | 256 |
| ***Reservoirs Non-linearity*** | tanh |
| ***Output vector units*** | 256 |
| ***Output vector distribution*** | bin.[-1,1] |
| ***Reservoirs states in readout*** | Vertically stacked |
| $\boldsymbol{W}_{in}^{(1)},\ \boldsymbol{W}_{in}^{(2)}$ | Uniform distribution of fp64 between -1 and 1 |
| $\rho(\boldsymbol{W}^{(1)}),\ \rho(\boldsymbol{W}^{(2)})$ | 0.95 |
| *Connectivity of reservoirs* | 100% |

* the input vectors are obtained flattening the blocks

Table 7: Baseline BBS-ESN weights and memory occupation

| ***Position*** | ***Number*** | ***Type*** | ***Occupied memory[KBytes]*** | ***Memory Type*** |
|---|---|---|---|---|
| $\boldsymbol{W}_{in}^{(1)}$ | 65,536 | fp64 | 512 | Flash |
| $\boldsymbol{W}^{(1)}$ | 65,536 | fp64 | 512 | Flash |
| $\boldsymbol{W}_{in}^{(2)}$ | 65,536 | fp64 | 512 | Flash |
| $\boldsymbol{W}^{(2)}$ | 65,536 | fp64 | 512 | Flash |
| $\boldsymbol{W}_{out}$ | 131,072 | fp64 | 1,024 | RAM |
| **Total** | 393,216 | fp64 | 2,048<br>1,024 | Flash<br>RAM |

Table 8: Baseline BBS-ESN description

| ***Operation*** | ***Multiply*** | ***Accumulate*** | ***Non- linearity*** |
|---|---|---|---|
| $\boldsymbol{a}(t) = \boldsymbol{W}_{in}^{(1)}\boldsymbol{u}(t)$ | 65,536 (fp64) | 65,280 (fp64) | - |
| $\boldsymbol{b}(t) = \boldsymbol{W}^{(1)}\boldsymbol{x}^{(1)}(t-1)$ | 65,536 (fp64) | 65,280 (fp64) | - |
| $\boldsymbol{x}^{(1)}(t) = \tanh\big(\boldsymbol{a}(t) + \boldsymbol{b}(t)\big)$ | - | 256 (fp64) | tanh |
| $\boldsymbol{c}(t) = \boldsymbol{W}_{in}^{(2)}\boldsymbol{x}^{(1)}(t)$ | 65,536 (fp64) | 65,280 (fp64) | - |
| $\boldsymbol{d}(t) = \boldsymbol{W}^{(2)}\boldsymbol{x}^{(2)}(t-1)$ | 65,536 (fp64) | 65,280 (fp64) | - |
| $\boldsymbol{x}^{(2)}(t) = \tanh\big(\boldsymbol{c}(t) + \boldsymbol{d}(t)\big)$ | - | 256 (fp64) | tanh |
| $\boldsymbol{y}(t) = \boldsymbol{W}_{out}\boldsymbol{x}(t)^*$ | 131,072 (fp64) | 130,816 (fp64) | - |
| $\boldsymbol{o}(t) = \mathrm{sign}\big(\boldsymbol{y}(t)\big)$ | - | - | sign |
| **Total** | 393,216 (fp64) | 392,448 (fp64) | - |

* $\boldsymbol{x}(t) = \begin{bmatrix}\boldsymbol{x}^{(1)}(t) & \boldsymbol{x}^{(2)}(t)\end{bmatrix}^T$.

As a consequence the resulting matrix (Equation 22)

$$\tilde{\boldsymbol{W}}_{out} = \frac{\boldsymbol{W}_{out}}{\min(|W_{out\ i,j}|)} \tag{22}$$

Table 9: Operations performed for the readout training of the baseline BBS-ESN

| ***Position*** | ***Multiply*** | ***Accumulate*** | ***Division*** |
|---|---|---|---|
| $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{X}^T + \boldsymbol{I}$ | 20,971,520 (fp64) | 20,709,632 (fp64) | - |
| $\boldsymbol{A} = \boldsymbol{K}^{-1}$ | $\boldsymbol{O}(512^3)^*$ | | |
| $\boldsymbol{Z} = \boldsymbol{Y}_{target}\boldsymbol{X}^T$ | 10,485,760 (fp64) | 10,354,688 (fp64) | - |
| $\boldsymbol{W}_{out} = \boldsymbol{Z}\boldsymbol{A}$ | 67,108,864 (fp64) | 66,977,792 (fp64) | - |

* The number of operations necessary for the matrix inversion depends on the algorithm used. It is in the order of $512^3$, being 256 the number of rows and columns of $\boldsymbol{K}$.

will be composed of elements that, taken in modulus, are always greater or equal to 1. Therefore, the values $\tilde{\boldsymbol{y}}$, obtained multiplying the BBS-ESN state for the the rescaled readout $\tilde{\boldsymbol{W}}_{out}$, will be a scaled version of $\boldsymbol{y}$, obtained with the original readout $\boldsymbol{W}_{out}$ (Equation 20). This is expressed in formulas in Equation 23

$$\tilde{\boldsymbol{W}}_{out}\boldsymbol{x}(t) = \tilde{\boldsymbol{y}}(t) = \frac{\boldsymbol{y}(t)}{\min(|W_{out\ i,j}|)}. \tag{23}$$

Therefore, the sign extraction of $\tilde{\boldsymbol{y}}(t)$, from Equation 21, would produce the same results of $\boldsymbol{o}(t)$ obtained with the baseline BBS-ESN, extractiong the sign from $\boldsymbol{y}(t)$. The elements of $\tilde{\boldsymbol{W}}_{out}$ are subsequently rescaled to *int8* representation introducing, inevitably, a quantization error. The procedure for rescaling the $\tilde{\boldsymbol{W}}_{out}$ values and converting them to *int8* numbers is reported in Equation 24

$$\tilde{\boldsymbol{W}}_{out}^{\text{int8}} = (int8)\Big(\frac{\tilde{\boldsymbol{W}}_{out}}{\max(|W_{out\ i,j}|)} \cdot (2^7 - 1)\Big). \tag{24}$$

Since the output of $\tilde{\boldsymbol{W}}_{out}/\max(|W_{out\ i,j|})$ is at most 1, the $\tilde{\boldsymbol{W}}_{out}^{\text{int8}}$ matrix coefficients will be at most equal to $2^7 - 1$.

The rescaling is the only additional training operation that is not executed for the baseline BBS-ESN. It is needed in order to quantize the readout weights as explained in the subsection 6.4.

The accumulations are performed in *int8*, even though the theoretical accumulation maximum could be overflowed. The reason is that the uniform distribution of binary values in the input matrices permits to never reach the accumulation maximums. This has been tested changing the accumulation type, in the training procedure, from *int8* to *int16* and the result in Table 15 showed no changes.

The quantized BBS-ESN operations and properties are summarized in Table 10, Table 11 and Table 12. As for the baseline BBS-ESN, in Table 11 is reported that the matrices $\boldsymbol{W}_{in}^{(1)}$ and $\boldsymbol{W}_{in}^{(2)}$ are to be be saved in Flash since they are constant like in the baseline implementation.

Table 10: Quantized BBS-ESN parameters

| | |
|---|---|
| ***Blocks size*** | $16 \times 16$ pixels |
| ***Blocks values distribution*** | bin.[-1, 1] |
| ***Input vector units*** | $256 \times 1$ pixels* |
| ***Number of reservoirs*** | 2 |
| ***Neurons per reservoirs*** | 256 |
| ***Reservoirs Non-linearity*** | sign |
| ***Output vector units*** | 256 |
| ***Output vector distribution*** | bin.[-1,1] |
| ***Reservoirs states in readout*** | Vertically stacked |
| $\boldsymbol{W}_{in}^{(1)}$, $\boldsymbol{W}_{in}^{(2)}$ | Uniform distribution of binary -1 and 1 |
| $\rho(\boldsymbol{W}^{(1)})$, $\rho(\boldsymbol{W}^{(2)})$ | 1.0 |
| ***Connectivity of reservoirs*** | 0.39% |

* the input vectors are obtained flattening the blocks

Table 11: Quantized BBS-ESN weights and memory occupation.

| ***Position*** | ***Number*** | ***Type*** | ***Occupied memory[KBytes]*** | ***Memory type*** |
|---|---|---|---|---|
| $\boldsymbol{W}_{in}^{(1)}$ | 65,536 | bin.[-1, 1] | 8 | Flash |
| $\boldsymbol{W}_{in}^{(2)}$ | 65,536 | bin.[-1, 1] | 8 | Flash |
| $\boldsymbol{W}_{out}$ | 131,072 | int8 | 128 | RAM |
| **Total** | 131,072<br>131,072 | bin.[-1, 1]<br>int8 | 16<br>128 | Flash<br>RAM |

* The multiplication of the reservoirs state for the reservoirs matrices are not reported, since those matrices have been set to be identity matrices.

Table 12: Quantized BBS-ESN inference operations for each block

| *Operation* | *Multiply* | *Accumulate* | *Non- linearity* |
|---|---|---|---|
| $\boldsymbol{a}(t) = \boldsymbol{W}_{in}^{(1)}\boldsymbol{u}(t)$ | 65,536 (bin.[-1, 1]) | 65,280 (int8) | - |
| $\boldsymbol{x}^{(1)}(t) = \text{sign}\big(\boldsymbol{a}(t) + \boldsymbol{x}^{(1)}(t-1)\big)$ | - | 256 (int8) | sign |
| $\boldsymbol{c}(t) = \boldsymbol{W}_{in}^{(2)}\boldsymbol{x}^{(1)}(t)$ | 65,536 (bin.[-1, 1]) | 65,280 (int8) | - |
| $\boldsymbol{x}^{(2)}(t) = \text{sign}\big(\boldsymbol{c}(t) + \boldsymbol{x}^{(2)}(t-1)\big)$ | - | 256 (int8) | sign |
| $\boldsymbol{y}(t) = \boldsymbol{W}_{out}\boldsymbol{x}(t)^*$ | 131,072 (int8) | 130,816 (int16) | - |
| $\boldsymbol{o}(t) = \text{sign}\big(\boldsymbol{y}(t)\big)$ | - | - | sign |
| **Total** | 131,072 (bin.[-1, 1])<br>131,072 (int8) | 131,072 (int8)<br>130,816 (int16) | - |

* $\boldsymbol{x}(t) = \big[\boldsymbol{x}^{(1)}(t) \quad \boldsymbol{x}^{(2)}(t)\big]^T$.

** The multiplication of the reservoirs state for the reservoirs matrices are not reported, since those matrices have been set to be identity matrices.

### 6.4.4 Memory savings

The weights memory occupation for the baseline BBS-ESN is of 2,048 KBytes of Flash and 1,024 KBytes of RAM, while for quantized BBS-ESN these values are 16 KBytes of Flash and 128 KBytes of RAM. The quantized network weights, consequently, occupies only the 0.8% of the baseline Flash memory and only the 12.5% of the baseline RAM. Section 10 reports an analysis on the implementation times and operations for the use of the quantized BBS-ESN.

Table 13: Operations performed for the readout training of the quantized BBS-ESN

| ***Position*** | ***Multiply*** | ***Accumulate*** | ***Division*** |
|---|---|---|---|
| $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{X}^T + \boldsymbol{I}$ | 20,971,520 (bin.[-1, 1]) | 20,709,632 (int8) | - |
| $\boldsymbol{A} = \boldsymbol{K}^{-1}$ | $\boldsymbol{O}(512^3)^*$ | | |
| $\boldsymbol{Z} = \boldsymbol{Y}_{target}\boldsymbol{X}^T$ | 10,485,760 (bin.[-1, 1]) | 10,354,688 (int8) | - |
| $\boldsymbol{W}_{out} = \boldsymbol{Z}\boldsymbol{A}$ | 67,108,864 (fp32 x int8) | 66,977,792 (fp32 + int8) | - |
| ***Rescale*** $\boldsymbol{W}_{out}$ ***to int8***$^{**}$ | - | - | 131.072 (fp32 / fp32) |

$^*$ The number of operations necessary for the matrix inversion depends on the algorithm used. It is in the order of $512^3$, being 256 the number of rows and columns of $\boldsymbol{K}$.

$^{**}$ The rescaling operation needs to be performed only for the quantized network

### 6.4.5 Quantized BBS-ESN readout training implementation

The quantized BBS-ESN training implements the operations reported in Table 13, using the formula from Equation 9, with $\alpha^2 = 1$. The matrix inversion in Table 13 is performed using *fp32* numbers. As a consequence, also the multiplication involving these values use *fp32*. Also for the case of the quantized BBS-ESN, in Table 13 the $\boldsymbol{K}$ matrix inversion computations are not reported, since it can be implemented with many methods. It is only reported the order of magnitude of the operations, in relation to the matrix dimension. Section 10 and Table 17 report the $\boldsymbol{K}$ inversion implementative details.

# 7 Method for anomaly detection

Anomalies are identified using a decision process called Evaluator. As already stated, the anomaly recognition is based on the evaluation of the reconstruction error between the BBS-ESN input blocks and output blocks. Specifically, the error that is taken into consideration is the number of pixel that is reproduced wrongly (i.e. the with the opposite binary value) comparing the network output $\boldsymbol{o}(t)$ and to the

corresponding input $\boldsymbol{u}(t)$ per block basis.

The Evaluator is trained with the error values from a number of images (considered normal) processed by the BBS-ESN, once its readout training has already been completed.

Three types of Evaluators have bee taken into account. In each Evaluator a number of images are processed by the network after the readout training. These images are numbered $\in \{T_1, \ldots, T_j, \ldots, T_n\}$ ($T$ stays for training). In the following notation, the images evaluated in inference will be named $I_1, \ldots, I_j, \ldots$, where $I$ stays for inference.

## 7.1 Evaluator on blocks with interception region

For this Evaluator it is computed, for each of the training images $\in \{T_1, \ldots, T_j, \ldots, T_n\}$, the mean of the wrongly reconstructed pixels per block, $\mu_{T_j}$, and the variance of the wrongly reconstructed pixels per block, $\sigma_{T_j}$.

A window of values, named training window, is subsequently set using these values. Its mean value $\bar{\mu_T}$ is the mean of the $\mu_{T_j}$, while its width is determined by $\bar{\sigma_T}$, that is the mean of the $\sigma_{T_j}$, multiplied by a $\Gamma$ coefficient. The training window will be constituted, therefore, by the values in Equation 25.

$$\text{Training window}_{\text{eval. on blocks}} = \left[\bar{\mu_T} - \Gamma\,\bar{\sigma_T},\ \bar{\mu_T} + \Gamma\,\bar{\sigma_T}\right]. \tag{25}$$

Once the values $\bar{\mu_T}$ and $\bar{\sigma_T}$ have been calculated, the Evaluator training is complete. For each new image processed in inference, numbered $I_1, \ldots, I_j, \ldots$, they are calculated the mean of the blocks error $\mu_{I_j}$ and the variance of the blocks error $\sigma_{I_j}$, with which it is calculated the widow of values corresponding to the image $I_j$. For an image to be considered normal, there must be a region of values that is common with all the other window of values from the preceding images, comprehending also the training window (Figure 17).

Since high noise percentages may alter the reconstruction of the images, this Evaluator has been designed to have a decision technique that allows the reconstructing values to have large variation margin. As will be seen in subsection 9.1, this Evaluator is not the best choice. In fact the variance of the reconstruction error $\sigma_{I_j}$ tends to enlarge in anomalous images, since they have normal blocks and anomalous blocks, being the anomaly located in only some blocks. High $\sigma_{I_j}$ enlarge the images windows of values, and this facilitates anomalous images to be evaluated as normal.

## 7.2 Evaluator on blocks without interception region

This type of Evaluator takes into account the same values of the Evaluator on blocks with interception region, but the interception region is not used. For an image to be considered normal there must be common values between the window of an image

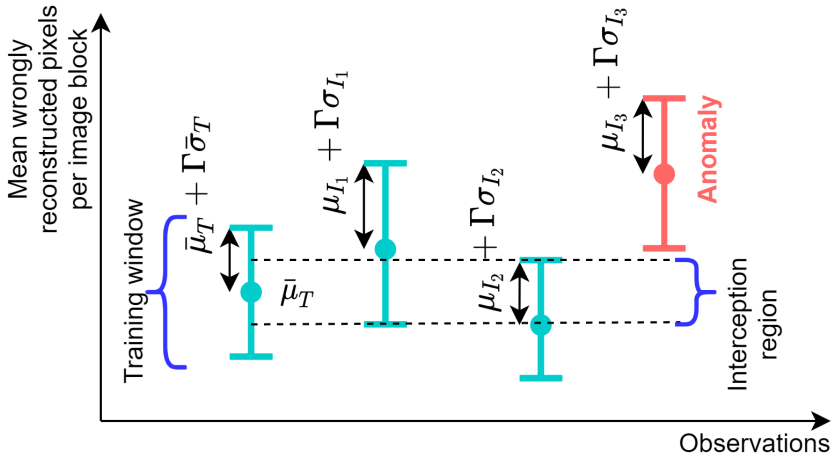



Figure 17: Detection of images as normal or anomalous using the Evaluator on blocks with interception region

processed in inference, $I_j$, and the training window (Figure 18).

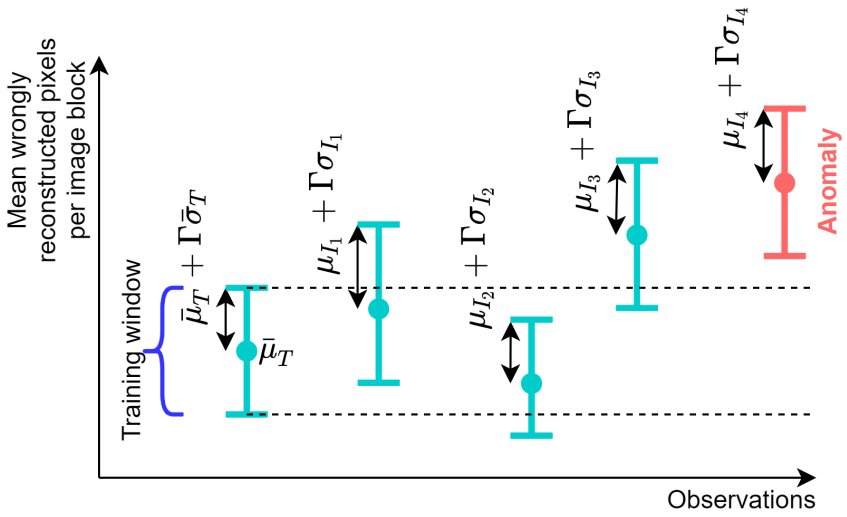



Figure 18: Detection of images as normal or anomalous using the Evaluator on blocks without interception region

This Evaluator, also, is not the best choice (subsection 9.1). Here can be found the same problem with $\sigma_{I_j}$ of the Evaluator with interception region, although it is mitigated by not using the interception region.

### 7.3 Evaluator on images

It can already be introduced that this is the Evaluator that has performed better results, as explained in subsection 9.1.

As the two Evaluators above, for each of the training images $\in \{T_1, \ldots, T_j, \ldots, T_n\}$,

the mean of the wrongly reconstructed pixels per block, $\mu_{T_j}$, is calculated. The mean of the $\mu_{T_j}$, named $\bar{\mu_T}$, is used also in this case as mean value of the Training window. The width of the window, in this case, is determined by the variance on the $\mu_{T_j}$, named $\sigma_T$, multiplied for a $\Gamma$ coefficient.

For this kind of Evaluator the width depends, apart from $\Gamma$, from the variance on the mean error of the training images, not on the mean of the variances of the error per block of the training images. This is the reason why this Evaluator is called "Evaluator on images".

The Training window, in this case, will be the one represented in Equation 26

$$\text{Training window}_{\text{eval. on image}} = \left[\bar{\mu_T} - \Gamma\,\sigma_T,\ \bar{\mu_T} + \Gamma\,\sigma_T\right]. \tag{26}$$

The rationale, therefore, is to perform the evaluation as the sampling from a pool of values, that can be normal or anomalous. This Evaluator has been designed in order to be simple (hence not computationally complex) and not to be biased by the reconstructing error variance in the images. A scheme of the Evaluator on image functioning is reported in Figure 19

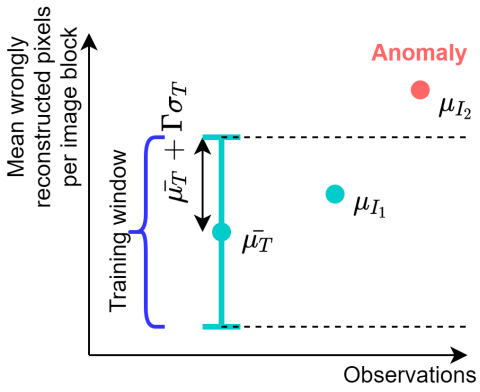



Figure 19: Detection of images as normal or anomalous using the Evaluator on imgae

## 7.4 Choice of $\Gamma$ coefficient

In order to reason about the choice of $\Gamma$, the accuracy metrics must be introduced first. The accuracy results that have been chosen in this study are the False Negative Ratio (FPR) and the False Positive Ratio (FNR). FPR in a sequence of images is defined as in Equation 27:

$$\text{FPR} = \frac{\text{False positives}}{\text{Images in the sequence}} \cdot 100, \tag{27}$$

where "False positives" is the number of normal images evaluated as anomalous.

FNR is defined as in Equation 28:

$$\text{FNR} = \frac{\text{False negatives}}{\text{Images in the sequence}} \cdot 100, \tag{28}$$

where "False negatives" is the number of anomalous images evaluated as normal.

The choice of the best $\Gamma$ coefficient depends on the type of Evaluator used and it is different for each noise level added to the sequences (to simulate defective phototransistors). In this study, the best $\Gamma$ values have been chosen a-posteriori, and they are the values that would be used in deployement mode. In fact, the choice of $\Gamma$ cannot be performed online. The reason is that the training can happen only with normal images, and the choice of $\Gamma$ depends on the distance of the windows (in the case of the Evaluaotors on blocks) or points (in the case of the Evaluator on images) between values corresponding to both normal and anomalous images.

The choice of $\Gamma$ is a trade-off. In fact, increasing it will lead to the inclusion of more values corresponding to anomalous images in the Training window (or Interception region), leading to an increment of the FNR. On the other hand, lowering $\Gamma$ will shrink the Training window, leaving out of it some values from normal images and increasing the FPR.

Both FPR and FNR can lead to negative effects. On the one hand the increasing of false negatives would cause the detection of anomalies only when they are prominent, causing delays in the maintenance service, at the risk of aggravating damages. On the other hand too many false positives could cause false alarms and useless maintenance interventions, leading to inefficiencies.

$\Gamma$ coefficients can be chosen in order to minimize FPR, FNR or a weighted sum of FPR and FNR, obtaining a trade-off between detection speed and false alarms quantity.

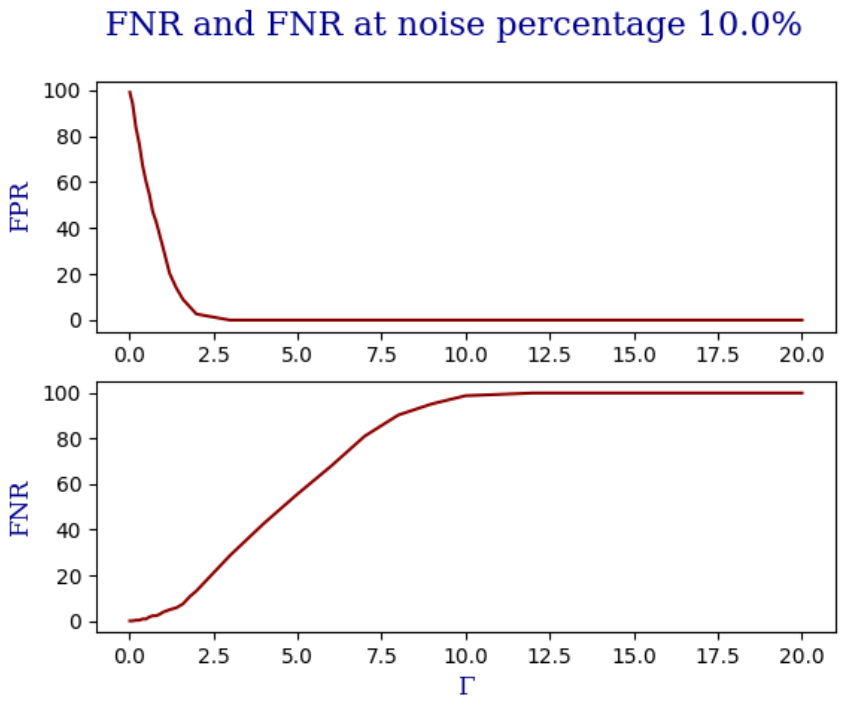



Figure 20: FPR and FNR varying Γ. Here the reconstruction took place the BBS-ESN and evaluation performed with Evaluator on images. The training and testing conditions are reported in section 8, using Lexicographical blocks order in the BBS-ESN training

As an exmaple, in Figure 20 they can be seen the FPR and FNR values for each of the Γ analyzed, with the Evaluator on Images, mediated over the 10 morphed sequences (section 5) with an addition of binary noise in 10% of the pixels in each image. The images reconstruction took place with the BBS-ESN after a training using blocks in Lexicographical order (section 8) and in the conditions explained in section 8.

It is useful to point out that more than one image are used in the training of the Evaluator because using a single image would cause the Training Window to be just a single value, treating as anomalous every image producing a different $\mu_{I_j}$ value. (section 7). Clearly, these condition would be acceptable only in ideal conditions, i.e. with total absence of defective photo-transistors. In fact, the binary noise that is added to the images in time-unrelated, making normal images in a sequence different between each other, while they would be all equal without noise, due to fixed framing.

# 8 Training and testing conditions

1. **Training blocks numbers:** In all the tests a fixed number of 80 blocks, collected from more than one image to achieve that quantity, have been used for

BBS-ESN training. Before using the states from those blocks for the training, the states from previous 50 blocks have been discarded, treating them as initial transient [7]. The choice of the training and transient blocks is explained in subsubsection 9.2.1.

2. **Training blocks order:** Two different input blocks orders have been used to construct the train matrices $\boldsymbol{X}$, with the states caused by the flattened input blocks, and $\boldsymbol{Y}_{target}$, composed by the target vectors. These target vectors are the flattened input blocks themselves, since the aim of the training is the reconstruction of the input blocks (Equation 17 and Equation 18).

   The possible orders for the input blocks are:

   - *Lexicographical order*: starting from the top left block, sliding right in the same row and repeating the process in the rows below. The process iterates for all the images used for the collection of training blocks.
   - *Random order*: the blocks order, once the blocks from all the images have been saved, is shuffled randomly. The pseudorandom number generation algorithm used for the random shuffle is PCG64 [58].

   This procedure has been done to test the sensibility of BBS-ESN to the input order. It is important to point out that the states in $\boldsymbol{X}$ follow the order of the input blocks in $\boldsymbol{Y}_{target}$, of which the states are a consequence.

3. **Number of images to train the Evaluator:** The Evaluator has been implemented using 100 normal images for its training, as a trade-off between test rapidity and accuracy results. In fact, the more images are used for the Evaluator training, the more the Training window mean and variance are computed using a wide samples population (that in this case are images).

4. $\Gamma$ **values used:** the $\Gamma$ values that have been used span from 0.02 to 20.

5. **Images in test sequences:** Once both the BBS-ESN and the Evaluator trainings are completed, sequences of 70 images have been used (the value 70 has been chosen as a trade-off between a significant number of images and a brief enough simulations time). In order to test the anomaly detection process in the most variable way, the number of normal images in each tested sequence (i.e. before the leak appearance) has been set differently (keeping the total images for each sequence to 70).

# 9 Analysis of the results

## 9.1 Choice of the best Evaluator

The three Evaluators have been tested with the baseline version of the BBS-ESN. All the 10 morphed sequences (section 5) have been tested with various noise levels.

For each of the noise levels used (0%, 0.001%, 0.01%, 0.1%, 1% and 10%) the results have been mediated over the 10 sequences.

The baseline BBS-ESN has been used for these tests because it ensures best precision and, consequently, an un-biased judgement on the best working Evaluator. In fact, variations on the results due to quantization may cause misleading decisions on the accuracy capabilities permitted by the Evaluators. The results obtained for each noise percentage and using each of the three Evaluators are reported in Table 14a.

Table 14: Evaluators accuracy results

(a) Best FPR and FNR results obtained for each Evaluator

| ***Evaluator type*** | ***Noise percentages*** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0% | | 0.001% | | 0.01% | | 0.1% | | 1% | | 10% | |
| Ev. on blocks with interc. region | 0.0 | 0.0 | 0.0 | 0.6 | 5.1 | 0.3 | 41.2 | 21.8 | 15.5 | 24.7 | 29.3 | 4.5 |
| Ev. on blocks without interc. region | 0.0 | 0.0 | 0.0 | 0.0 | 0.3 | 0.0 | 6.0 | 0.0 | 5.7 | 0.0 | 23.4 | 6.8 |
| Ev. on images | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 2.7 | 13.2 |

(b) Sum of FPR and FNR for all the noise percentages for each Evaluator

| ***Evaluator type*** | ***FPR + FNR summed for all noise percentages*** |
|---|---|
| Ev. on blocks with interc. region | 143.0 |
| Ev. on blocks without interc. region | 42.2 |
| Ev. on images | 15.9 |

The Evaluator on images (subsection 7.3) is the one that permits better accuracy performances because it is the one that has the smaller FPR + FNR value summed for all the noise percentages, as can be seen in Table 14b

## 9.2 Accuracy results with the baseline and quantized BBS-ESN, using the Evaluator on images

Once chosen the Evaluator on images as the one with better performances, the morphed sequences from section 5 have been processed with both the versions of the BBS-ESN. Table 15 shows the accuracy results. The comparison with the Residual Image Algorithm is explained in subsubsection 9.3.2. Figure 21 shows $\Gamma$ values that allow to achieve those results for each noise level. For the sake of brevity, only the values used to minimize the sum of FPR and FNR are reported.

Table 15: Accuracy results of the baseline and quantized BBS-ESN, compared with an anomaly detection algorithm that evaluates the residual image through image subtraction. A different Γ coefficient has been used for each network and each noise percentage, in order to minimize the FPR, the FNR and the sum of FPR and FNR.

| ***Minimization*** | ***Input-output comparison*** | ***BBS-ESN training blocks order*** | ***Noise percentages*** | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 0.1% | | 1% | | 10% | |
| | | | FPR | FNR | FPR | FNR | FPR | FNR |
| FPR | Baseline BBS-ESN | Random | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 28.7 |
| | | Lexicographical | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 28.7 |
| | Quantized BBS-ESN | Random | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 33.1 |
| | | Lexicographical | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 33.1 |
| | Residual Image | - | 0.0 | 22.7 | 0.0 | 54.4 | 0.0 | 92.5 |
| FNR | Baseline BBS-ESN | Random | 0.0 | 0.0 | 0.0 | 0.0 | 83.8 | 0.3 |
| | | Lexicographical | 0.0 | 0.0 | 0.0 | 0.0 | 83.8 | 0.3 |
| | Quantized BBS-ESN | Random | 0.0 | 0.0 | 0.0 | 0.0 | 81.4 | 0.6 |
| | | Lexicographical | 0.0 | 0.0 | 0.0 | 0.0 | 81.4 | 0.6 |
| | Residual Image | - | 32.1 | 0.0 | 83.8 | 5.0 | 85.4 | 9.9 |
| FPR + FNR | Baseline BBS-ESN | Random | 0.0 | 0.0 | 0.0 | 0.0 | 2.7 | 13.2 |
| | | Lexicographical | 0.0 | 0.0 | 0.0 | 0.0 | 2.7 | 13.2 |
| | Quantized BBS-ESN | Randon | 0.0 | 0.0 | 0.0 | 0.0 | 4.1 | 14.6 |
| | | Lexicographical | 0.0 | 0.0 | 0.0 | 0.0 | 4.1 | 14.6 |
| | Residual Image | - | 1.2 | 5.6 | 6.6 | 28.9 | 30.7 | 38.6 |

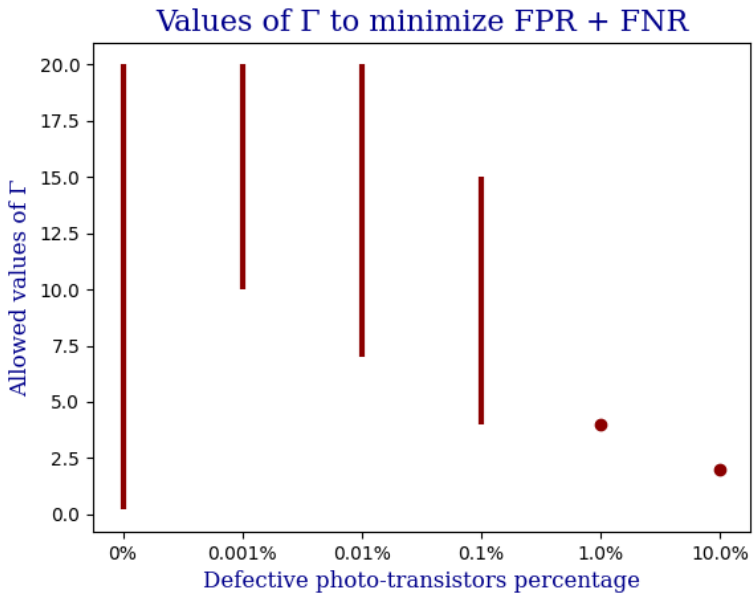
Results with defective pixels percentage of 0%, 0.001% and 0.01% have not been reported since they are always FPR = 0.0, FNR = 0.0.
The number transient blocks is 50 for each BBS-ESN test.
The number of blocks to train the readout is 80 for each BBS-ESN test.
The number of images used to train the Evaluator is 100 in every case.

### 9.2.1 Choice of the number of training and transient blocks

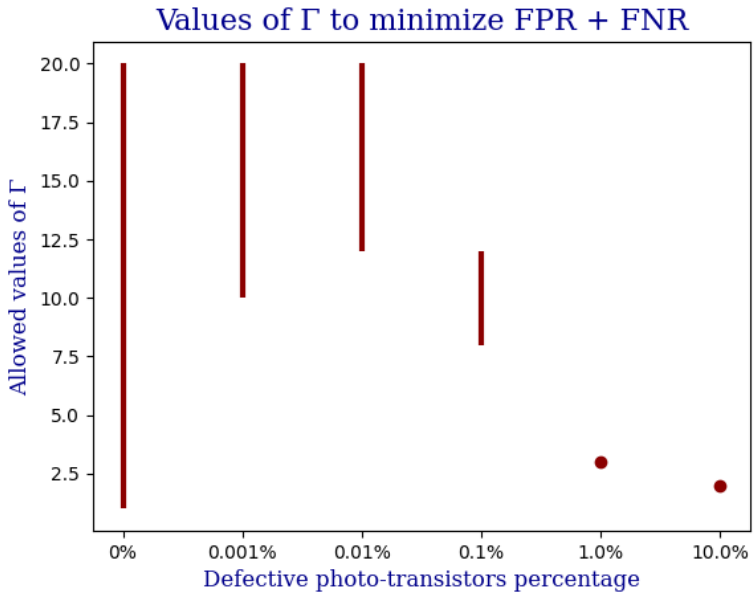
For the choice of the number of training blocks, various blocks number have been tested (200, 100, 90, 80, 60, 30) with both the baseline and quantized BBS-ESN with a fixed transient blocks of 50 (that is approximately the number of blocks in two images for the images with a smaller blocks number, which has a mean of 29.4). 80 has been the only number that permitted FPR = 0.0 and FNR = 0.0 at the noise level of 1% for both the BBS-ESN versions. As for the number of transient blocks, various values have been tested (100, 80, 50, 30, 0) using both the BBS-ESN versions, trained with 80 blocks. 50, as number of transient blocks, has been the only value allowing FPR = 0.0, FNR = 0.0 at noise level 1% for both the baseline and quantized BBS-ESN.

## 9.3 Analysis of the results

As the noise increases it becomes increasingly difficult for the network to separate the $\mu_{I_j}$ values corresponding to normal and anomalous images (section 7), using the




(a) Gamma coefficients for the quantized BBS-ESN allowing the results obtained.




(b) Gamma coefficients for the quantized BBS-ESN allowing the results obtained.

Figure 21: Gamma values allowing the results obtained. The values are the same for both the blocks order used for the network training (i.e. lexicographical and random).

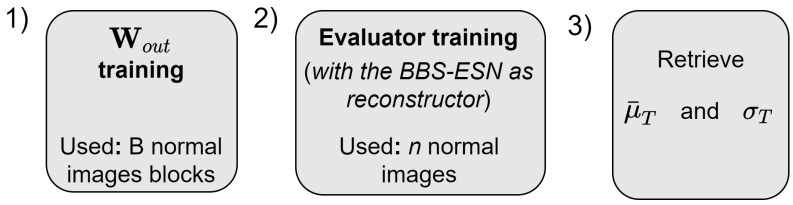
Evaluator on images. This is why there is no $\Gamma$ permitting to obtain FPR $= 0.0$ and FNR $= 0.0$ at a noise percentage of 10%.

### 9.3.1 Robustness to training blocks order

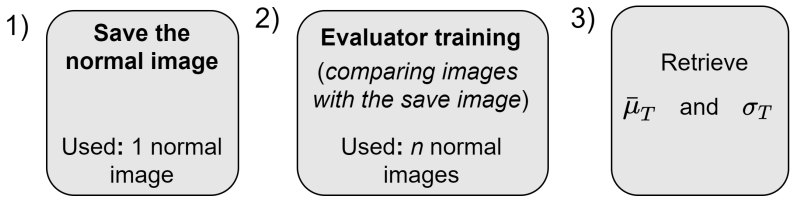
Following the results of the experiments it can be stated that both the BBS-ESN versions are robust to changing in the blocks order for the training of the readout matrix $\boldsymbol{W}_{out}$. In fact, the results obtained are the same (at one decimal place in FPR and FNR, as reported in Table 15) in both the train condition.

### 9.3.2 Comparison with Residual Image Algorithm

The results have been compared with the ones achieved by using a Residual Image Algorithm, that compares a newly acquired images with a saved one that is considered normal, like the method in [59]. For this algorithm the Evaluator on images that has been used follows the same rules of the Evaluator on images used for the BBS-ESN. Figure 22 compares the entire pipeline of input-output comparison method, using the BBS-ESN or the Residual Image Algorithm.




(a) Pipeline for BBS-ESN trainining and Evaluator on images training




(b) Pipeline for Residual Image Algorithm image acquisition and Evaluator on images training

Figure 22: Comparison of the pipelines for the usage of the BBS-ESN and the Residual Image Algorithm for input-output comparison.

Both BBS-ESN, baseline and quantized versions, have achieved better results than the Residual Image Algorithm. This is due to a greater reconstruction error that the BBS-ESN is capable to achieve for anomalous images and to a lower reconstruction error for normal images. This leads to a better separation between the $\mu_{I_j}$ from normal and anomalous images (subsection 7.3), leading to overall better performances in terms of FPR and FNR.

## 9.4 Further tests

### 9.4.1 Test with different quantization procedure

Experiments have been conducted with the training procedure of the quantized BBS-ESN, in the attempt to reduce the RAM requirements and the computational complexity. It has been rescaled the matrix $\boldsymbol{K}^{-1}$ (Table 13) and converted it to integer representation before the multiplication with $\boldsymbol{Z}$, with the aim of reducing the times required for executing the multiplication (the chosen representation has been *int32*, to reduce the overflow possibility). The results, using lexicographical training blocks order and choosing $\Gamma$ coefficients in order to minimize the sum of FPR + FNR, scored FPR = 0.8, FNR = 1.9 at a noise level of 1% and FPR = 3.2, FNR = 13.1 at a noise level of 10%. Due to the accuracy reduction towards the original method reported in Table 13, the original procedure has been kept. In fact, this new method would cause a performance reduction at noise level 1%, since using this noise level with the original quantization procedure we have FPR = 0.0, FNR = 0.0.

### 9.4.2 Removal of the recurrent connections

Experiments have been conducted removing the recurrent connections with the baseline and quantized BBS-ESNs, reducing the network to be feedforward: a classic ELM (Section 3.6) with 2 hidden layers instead of 1. This also means that the elements $\boldsymbol{W}^{(1)}\boldsymbol{x}^{(1)}(t-1)$ and $\boldsymbol{W}^{(2)}\boldsymbol{x}^{(2)}(t-1)$ of equation 11 have been equaled to 0. Consequently, the operations for computing $\boldsymbol{x}^{(1)}(t)$ and $\boldsymbol{x}^{(2)}(t)$ in Table 8 and Table 12 are transformed to the sole application of the element-wise sign extraction.

Results (not shown in Table 15) show 0.0 values for both FPR and FNR in the using sequences with 0% noise, for both the baseline and quantized feedforward networks and for both the lexicographical and random training blocks orders.

As for the conditions of 1% and 10% noise percentages, the baseline feeforward network, trained with lexicographical blocks order, scored results of FPR = 0.3, FNR = 0.3 for 1% noise and FPR = 4.5, FNR = 12.3 for 10% noise. Using a random blocks order the accuracy results were FPR = 0.3, FNR = 0.0 for 1% noise and FPR = 3.2, FNR = 13.0 for 10% noise.

The quantized feedforward network scored results of FPR = 0.3, FNR = 0.3 for 1% noise and and FPR = 4.1, FNR = 11.7 for 10% noise with a lexicographical training order. The results for the random training order with noise levels of 1% and 10% were, respectively, FPR = 0.3, FNR = 0.0 and FPR = 3.9, FNR = 12.0.

These results were obtained using Gamma coefficient for the minimization of the sum of FPR and FNR.

Results obtained by the BBS-ESN baseline and quantized versions are marginally different. For noise level 1% the difference is under 1% for both FPR and FNR for both the baseline and quantized versions. For noise level 10% the difference is under

1% for both FPR and FNR for the baseline version and under 1% for FPR and under 3% for FNR for the quantized version.

# 10 Feasibility analysis on a tiny micro controller

The execution times of the quantized BBS-ENS version have been estimated considering the implementation one the micro-controller STM32H743ZI2 running at 480 MHz, featuring ARM Cortex-M7 core, 1 MByte embedded RAM and 2 MByte embedded FLASH. The baseline BBS-ESN is out of profiling scope due to the usage of double precision floatin point (*fp64*) numbers that reduces time and memory performances.

By profiling a variety of neural layers at various bit-depth, generated with the plugin X-CUBE-AI v7.0.0 part of STM32CubeMX [60], it has been calculated that by average the execution time of a single precision floating point (*fp32*) MACC is 10.4 ns. For a binary MACC it is 1.5 ns, for a *int8* MACC it is 6 ns and, finally, for a *fp32* division it is 29 ns. On the basis of these values, they can be estimated the execution times required for training and inference of the quantized BBS-ESN.

Being possible to implement up to 4 *int8* sums in one clock cycle (using ARM SIMD instructions [61]), the time for an *int8* sum can be approximated to be $\frac{1/4}{480\text{ MHz}} = 0.52$ ns. In a similar way, 2 *int16* sums can be performed in 1 clock cycle in an estimated time of $\frac{1/2}{480\text{ MHz}} = 1.04$ ns.

In this study, binary values (-1 and 1) are packed as 0 and 1 in 32 bit words but they are expressed as -1 and 1 in *int8* values, in order to perform computations. The time required for packing and unpacking these values to and from 32 bit words has not been considered. It is important to notice, also, that in these estimates the operations and the times for reading and writing into the registers are not considered. The overall analysis, as a consequence, is an optimistic one.

In Table 16 are reported the times for the operations spent during training and inference. The necessary time for a binary multiplication has been approximated with the time required for a binary MACC.

Table 16: Times estimated for the single operations spent for training and inference

| ***Operation*** | ***Time[ms]*** |
|---|---|
| int8 sum | $0.52 \cdot 10^{-6}$ |
| int16 sum | $1.04 \cdot 10^{-6}$ |
| bin.[-1, 1] multiplication | $1.5 \cdot 10^{-6*}$ |
| fp32 MACC | $10.4 \cdot 10^{-6}$ |
| fp32 division | $29 \cdot 10^{-6}$ |

* The time used for the binary multiplication is approximated to the time for a binary MACC.

## 10.1 Training time profile

Table 17 reports the execution times and the number of operations required for the readout training. These operations are represented with the types used for the MCU implementation. This means that accumulations that, theoretically, would be of *fp32* with *int8* values, are implemented with both numbers in *fp32*).

The times for *fp32* operations represent a slightly worse case, since the time for *fp32* multiplications and sums is approximated with the time for a number of MACC equal to the multiplications number, that is slightly greater than the number of sums. For the estimate of the matrix inversion time, the Gauss-Jordan method has been considered, being the method used for the inversion in the ARM CMSIS-5 library [62].

Using 100 images to be trained (section 8), the Evaluator would require a training time of 1.206 s.

Table 17: Times required and respective operations for the readout training for the quantized BBS-ESN. The time for converting binary values, packed as 1-bit values, into *int8* has not been considered in this estimation.

| ***Position*** | ***Times[ms]*** | ***Multiply*** | ***Accumulate*** | ***Division*** |
|---|---|---|---|---|
| $\boldsymbol{K = XX^T + I}$ | $0.042 \cdot 10^3$ | 20,971,520 (bin.[-1, 1]) | 20,709,888 (int8) | - |
| $\boldsymbol{A = K^{-1}}$* | $0.467 \cdot 10^3$** | 44,869,888 (fp32) | 44,869,888 (fp32) | 131,328 (fp32) |
| $\boldsymbol{Z = Y_{target}X^T}$ | $0.021 \cdot 10^3$ | 10,485,760 (bin.[-1, 1]) | 10,354,688 (int8) | - |
| $\boldsymbol{W_{out} = ZA}$ | $0.698 \cdot 10^3$** | 67,108,864 (fp32) | 66,977,792 (fp32) | - |
| **Rescale** $\boldsymbol{W_{out}}$ | $0.004 \cdot 10^3$ | - | - | 131,072 (fp32) |
| **Total** | $1.232 \cdot 10^3$ | 31,457,280 (bin.[-1, 1]) 111,978,752 (fp32) | 31,064,576 (int8) 111,847,680 (fp32) | 262,400 (fp32) |

* The Gauss-Jordan method has been considered for the inversion.
** The time required for *fp32* multiplications and sums is obtained. approximating the *fp32* MACC quantity with the multiplication quantity

## 10.2 Inference time profile

The times and operations for a single block inference, using the quantized BBS-ESN, are listed in Table 18. The multiplication of the reservoirs states for the reservoirs matrices are not reported because those matrices are set to be identity matrices.

Since the estimated one block inference time is 0.402 ms and the mean blocks number per image in the sequences used is 29.4, then approximated to 30, the time required to perform the inference on one image is estimated with 12.06 ms. The reason for having different blocks number in different sequences is illustrated in

section 5. As for the mean computational complexity, the computation numbers must be multiplied by 30 to obtain the mean computations per image. They are estimated with 7,864,320 binary multiplications, 3,932,160 *int8* sums and 3,924,480 *int16* sums. The time required to compare the BBS-ESN output $\boldsymbol{o}(t)$ with the input $\boldsymbol{u}(t)$ has been neglected, being a simple XOR operation and a count of the 1 values in output from the XOR, of second order importance versus to the other operations. Furthermore, the computation of the mean number of 1 values per block in output from the XOR, that is the $\mu_{I_j}$ corresponding to the image (subsection 7.3), has been neglected for the same reason.

Table 18: Times required and respective operations for one block inference for the quantized BBS-ESN

| ***Operation*** | ***Time[ms]*** | ***Multiply*** | ***Accumulate*** | ***Non-lin.*** |
|---|---|---|---|---|
| $\boldsymbol{a}(t) = \boldsymbol{W}_{in}^{(1)}\boldsymbol{u}(t)$ | 0.034 | 65,536 (bin.[-1, 1]) | 65,280 (int8) | - |
| $\boldsymbol{x}^{(1)}(t) = \text{sign}\big(\boldsymbol{a}(t) + \boldsymbol{x}^{(1)}(t-1)\big)$ | 0.001* | - | 256(int8) | sign |
| $\boldsymbol{c}(t) = \boldsymbol{W}_{in}^{(2)}\boldsymbol{x}^{(1)}(t)$ | 0.034 | 65,536 (bin.[-1, 1]) | 65,280 (int8) | - |
| $\boldsymbol{x}^{(2)}(t) = \text{sign}\big(\boldsymbol{c}(t) + \boldsymbol{x}^{(2)}(t-1)\big)$ | 0.001* | - | 256(int8) | sign |
| $\boldsymbol{y}(t) = \boldsymbol{W}_{out}\boldsymbol{x}(t)$** | 0.333 | 131,072 (bin.[-1, 1]) | 130,816 (int16) | - |
| $\boldsymbol{o}(t) = \text{sign}\big(\boldsymbol{y}(t)\big)$*** | - | - | - | sign |
| **Total** | 0.402 | 262,144 (bin.[-1, 1]) | 131,072 (int8)<br>130,816 (int16) | - |

* The time required for the *int8* accumulation and sign extraction has been approximated with the time required for a binary multiplication and *int8* sum.

** $\boldsymbol{x}(t) = \begin{bmatrix}\boldsymbol{x}^{(1)}(t) & \boldsymbol{x}^{(2)}(t)\end{bmatrix}^T$

*** The times for the sign extraction of $\boldsymbol{y}(t)$ have been neglected, having overestimated the times for the *int8* sum and sign extraction (in the numbers with the * symbol).

The multiplication of the reservoirs state for the reservoirs matrices are not reported, since we have set those matrices to be equal to identity matrices.

## 10.3 Training required memory

The RAM required to perform the training (as described in Table 17) is reported in Table 19 with all the necessary steps to be executed. The maximum required RAM value is needed for the $\boldsymbol{K}$ matrix inversion and it is 2,176 KBytes. This quantity has been estimated in the worst optimization case (i.e. in the need of storing simultaneously every value of $\boldsymbol{K}$ and $\boldsymbol{K}^{-1}$, with the $\boldsymbol{K}$ coefficients expressed as *fp32* numbers).

Table 19: RAM required for the training of the BBS-ESN. $\boldsymbol{X}$ is the state matrix (Equation 17), $\boldsymbol{Y}_{target}$ is the target matrix (Equation 18).

| ***Step*** | ***Matrices needed*** | ***Dimension and type*** | ***Required RAM for matrix [KBytes]*** | ***Total required RAM [KBytes]*** |
|---|---|---|---|---|
| Before training | $\boldsymbol{X}$ | $512 \times 80$ (bin.[-1, 1]) | 5 | 7.5 |
| | $\boldsymbol{Y}_{target}$ | $256 \times 80$ (bin.[-1, 1]) | 2.5 | |
| $\boldsymbol{Z} = \boldsymbol{Y}_{target}\boldsymbol{X}^T$ | $\boldsymbol{X}$ | $512 \times 80$ (bin.[-1, 1]) | 5 | 135.5 |
| | $\boldsymbol{Y}_{target}$ | $256 \times 80$ (bin.[-1, 1]) | 2.5 | |
| | $\boldsymbol{Z}$ | $256 \times 512$ (int8) | 128 | |
| $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{X}^T + \boldsymbol{I}$ | $\boldsymbol{X}$ | $512 \times 80$ (bin.[-1, 1]) | 5 | 389 |
| | $\boldsymbol{Z}$ | $256 \times 512$ (int8) | 128 | |
| | $\boldsymbol{K}$ | $512 \times 512$ (int8) | 256 | |
| $\boldsymbol{K}^{-1}$* | $\boldsymbol{K}$ | $512 \times 512$ (fp32) | 1,024 | 2,176 |
| | $\boldsymbol{K}^{-1}$ | $512 \times 512$ (fp32) | 1,024 | |
| | $\boldsymbol{Z}$ | $256 \times 512$ (int8) | 128 | |
| $\boldsymbol{Z}\boldsymbol{K}^{-1} = \boldsymbol{W}_{out}$ | $\boldsymbol{K}^{-1}$ | $512 \times 512$ (fp32) | 1,024 | 1,664 |
| | $\boldsymbol{Z}$ | $256 \times 512$ (int8)** | 128 | |
| | $\boldsymbol{Z}\boldsymbol{K}^{-1}$ | $256 \times 512$ (fp32) | 512 | |
| Rescaling of $\boldsymbol{W}_{out}$ | $\boldsymbol{W}_{out}$ | $256 \times 512$ (fp32) | 512 | 512 |
| Conversion of $\boldsymbol{W}_{out}$ in int8 | $\boldsymbol{W}_{out}$ | $256 \times 512$ (fp32) | 512 | 512*** |
| | $\boldsymbol{W}_{out}$ | $256 \times 512$ (int8) | 128 | |
| Final result: $\boldsymbol{W}_{out}$ in int8 | $\boldsymbol{W}_{out}$ | $256 \times 512$ (int8) | 128 | 128 |
| - | - | - | Maximum required RAM | 2,176 |

* the RAM for the matrix inversion has been estimated in the worst optimization case (i.e. in the necessity of storing simultaneously every value of $\boldsymbol{K}$ and $\boldsymbol{K}^{-1}$, with $\boldsymbol{K}$ converted in *fp32*).
** $\boldsymbol{Z}$ can be saved in *int8* and every element converted to *fp32* before multiplying it.
*** 512 KBytes is the maximum RAM required in this step because the elements of $\boldsymbol{W}_{out}$ can be converted to *int8* overwriting the *fp32* values.

## 10.4 Inference required memory

By implementing the inference with the operations reported in Table 20, then the total RAM required for one block inference is 129.2 KBytes, considering the readout matrix $\boldsymbol{W}_{out}$ and the input, state and output, respectively $\boldsymbol{u}(t)$, $\boldsymbol{x}(t)$ and $\boldsymbol{y}(t)$. These values are shown in Table 20. As for the Flash memory, the required quantity for one block inference is 16 KBytes, as reported in Table 11.

Table 20: RAM required to perform the inference with the operations of Table 18.

| ***Matrix or vector*** | ***Dimension and type*** | ***Required RAM[KBytes]*** |
|---|---|---|
| $\boldsymbol{W}_{out}$ | $256 \times 512$ (int8) | 128 |
| $\boldsymbol{u}(t)^*$ | $256 \times 1$ (bin.[-1, 1]) | 0.03 |
| $\boldsymbol{x}(t)^*$ | $512 \times 1$ (bin.[-1, 1]) | 0.06 |
| $\boldsymbol{o}(t)^*$ | $256 \times 1$ (bin.[-1, 1]) | 0.03 |
| - | Total required RAM | 129.2 |

* The elements of $\boldsymbol{u}(t)$, $\boldsymbol{x}(t)$ and $\boldsymbol{o}(t)$ can be saved as binary (0 and 1) in 32bit words and converted to *int8* (-1 and 1) to perform operations, one value at a time.

# 11 Conclusions and future works

This study proved that the quantized BBS-ESN can be implemented in inference with an estimated number of operations, per image, of 7,864,320 binary multiplications, 3,932,160 *int8* sums and 3,924,480 *int16* sums, and with an execution time of 12.06 ms. The estimates on the required RAM and Flash are respectively of 129.2 KBytes and 16 KBytes.

As for the quantized BBS-ESN training, the estimated numbers and types of operations are 31,457,280 binary multiplications, 111,978,752 *fp32* multiplications, 31,064,576 *int8* sums and 111,847,680 *fp32* sums, requiring an execution time of 1.232 s. For the training it is not required any additional space in Flash and the required RAM is estimated to be 2,176 KBytes. Once the network has been trained, the decision process called Evaluator trains to set the reconstruction error threshold, to separate normal images from anomalous images, in 1.260 s (subsection 10.2).

The accuracy results that have been obtained with the quantized BBS-ESN are of FPR = 0.0, FNR = 0.0 in the less challenging noise conditions (0%) and of FPR = 4.1, FNR = 14.6 in the most challenging noise conditions (10%). These are the best results for the sum of FPR and FNR and they are achieved setting the Evaluator in order to minimize this sum. In the same conditions the baseline BBS-ESN has performances of FPR = 0.0, FNR = 0.0 with no noise and FPR = 2.7, FNR = 13.2 with a noise percentage of 10%.

The Flash memory saving for inference, between the baseline and the quatized BBS-ESN, is of 2,032 KBytes, comparing the value of Table 11 with the value of Table 7

The RAM saving for inference, in terms of required weights memory, is of 896 KBytes, comparing the value of Table 11 to the value of Table 7.

It is important to note, as stated in Section 10, that the estimates are optimistic, since the operations and the time for reading and writing the data in the registers are not considered. Moreover, the time and the operations required to convert binary values into *int8*, and vice-versa (section 10) has been neglected. The reason for this is that these operations are of second order importance versus the others.

This study shown that the quantized BBS-ESN can be implemented on a off-the-shelf MCU, due to the deep quantization applied to the baseline BBS-ESN version. This is feasible because the estimated time for the inference on one image is approximately 12.06 ms (subsection 10.2) and the images could be acquired every minute or even less frequently, since the relatively slow process of oil leak motion. Slight accuracy differences have been noted between the BBS-ESN and the network maintaining the same architecture but without recurrent connections, both in the quantized and baseline versions (subsubsection 9.4.2).

The network training could be optimized using half precision floating point numbers, or even fixed point precision numbers. The training states could be reduced in smaller chunks making the training iterative between chunks until all of them are processed and the weights computed. This approach would be useful, because it is expected to reduce the RAM needs. Moreover, more networks architectures could be tested, changing the combinations of reservoir layers, neurons per reservoir and blocks dimensions. Finally, the network could be implemented in C-code, allowing to measure on the device the quantities that have been estimated.

# References


[1] *Renewable Capacity Statistics 2019*. Tech. rep. IRENA, 2019.

[2] *Renewable Energy Statistics 2018*. Tech. rep. IRENA, 2019.

[3] Magdi Mahmoud and Yuanqing Xia. "Some Industrial Systems". In: Jan. 2012, pp. 11–33.

[4] Yann LeCun, Y. Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44.

[5] Claudio Gallicchio, Alessio Micheli, and Luca Pedrelli. "Deep reservoir computing: A critical experimental analysis". In: *Neurocomputing* 268 (2017). Advances in artificial neural networks, machine learning and computational intelligence, pp. 87–99.

[6] Claudio Gallicchio and Alessio Micheli. "Deep Echo State Network (DeepESN): A Brief Survey". In: *CoRR* abs/1712.04323 (2017). arXiv: 1712.04323.

[7] Herbert Jaeger. "The" echo state" approach to analysing and training recurrent neural networks-with an erratum note'". In: *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report* 148 (Jan. 2001).

[8] Mantas Lukoševičius and Herbert Jaeger. "Reservoir computing approaches to recurrent neural network training". In: *Computer Science Review* 3.3 (2009), pp. 127–149.

[9] Denis Kleyko et al. *Integer Echo State Networks: Efficient Reservoir Computing for Digital Hardware*. 2020. arXiv: 1706.00280 [cs.NE].

[10] Claudio Gallicchio and Alessio Micheli. "Architectural and Markovian factors of echo state networks". In: *Neural Networks* 24.5 (2011), pp. 440–456.

[11] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly Detection: A Survey". In: *ACM Comput. Surv.* 41.3 (July 2009).

[12] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. "A survey on unsupervised outlier detection in high-dimensional numerical data". In: *Statistical Analysis and Data Mining: The ASA Data Science Journal* 5.5 (2012), pp. 363–387.

[13] Manish Gupta et al. "Outlier Detection for Temporal Data: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 26.9 (2014), pp. 2250–2267.

[14] Victoria Hodge and Jim Austin. "A survey of outlier detection methodologies". In: *Artificial intelligence review* 22.2 (2004), pp. 85–126.

[15] Hui Lu et al. "An Outlier Detection Algorithm Based on Cross-Correlation Analysis for Time Series Dataset". In: *IEEE Access* 6 (2018), pp. 53593–53610.

[16] Faraz Rasheed and Reda Alhajj. "A Framework for Periodic Outlier Pattern Detection in Time-Series Sequences". In: *IEEE Transactions on Cybernetics* 44.5 (2014), pp. 569–582.

[17] Shixiong Wang, Chongshou Li, and Andrew Lim. "A Model for Non-Stationary Time Series and its Applications in Filtering and Anomaly Detection". In: *IEEE Transactions on Instrumentation and Measurement* 70 (2021), pp. 1–11.

[18] Min Hu et al. "Detecting Anomalies in Time Series Data via a Meta-Feature Based Approach". In: *IEEE Access* 6 (2018), pp. 27760–27776.

[19] Hermine N. Akouemo and Richard J. Povinelli. "Data Improving in Time Series Using ARX and ANN Models". In: *IEEE Transactions on Power Systems* 32.5 (2017), pp. 3352–3359.

[20] Qing Chen et al. "Imbalanced dataset-based echo state networks for anomaly detection". In: *Neural Computing and Applications* 32 (Apr. 2020).

[21] Niklas Heim and James E. Avery. "Adaptive Anomaly Detection in Chaotic Time Series with a Spatially Aware Echo State Network". In: *CoRR* abs/1909.01709 (2019). arXiv: 1909.01709.

[22] Oliver Obst, X. Rosalind Wang, and Mikhail Prokopenko. "Using Echo State Networks for Anomaly Detection in Underground Coal Mines". In: *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*. 2008, pp. 219–229.

[23] Yıldız Karadayı, Mehmet N. Aydin, and A. Selçuk Öğrenci. "A Hybrid Deep Learning Framework for Unsupervised Anomaly Detection in Multivariate Spatio-Temporal Data". In: *Applied Sciences* 10.15 (2020).

[24] Suwon Suh et al. "Echo-state conditional variational autoencoder for anomaly detection". In: *2016 International Joint Conference on Neural Networks (IJCNN)*. 2016, pp. 1015–1022. DOI: 10.1109/IJCNN.2016.7727309.

[25] Jaemann Park et al. "utilizing online learning based on echo-state networks for the control of a hydraulic excavator". In: *Mechatronics* 24.8 (2014), pp. 986–1000.

[26] Tim Waegeman, Francis wyffels, and Benjamin Schrauwen. "Feedback Control by Online Learning an Inverse Model". In: *IEEE Transactions on Neural Networks and Learning Systems* 23.10 (2012), pp. 1637–1648.

[27] Jean P. Jordanou, Eric Aislan Antonelo, and Eduardo Camponogara. "Online learning control with Echo State Networks of an oil production platform". In: *Engineering Applications of Artificial Intelligence* 85 (2019), pp. 214–228.

[28] Voulodimos A et al. "Online classification of visual tasks for industrial workflow monitoring". In: *Neural Netw. 2011 Oct* (2011). DOI: 10.1016/j.neunet.2011.06.001.

[29] Kexin Xing et al. “Modeling and control of McKibben artificial muscle enhanced with echo state networks”. In: *Control Engineering Practice* 20.5 (2012), pp. 477–488. ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2012.01.002. URL: https://www.sciencedirect.com/science/article/pii/S0967066112000032.

[30] Kan Zeng and Yixiao Wang. “A Deep Convolutional Neural Network for Oil Spill Detection from Spaceborne SAR Images”. In: *Remote Sensing* 12 (Mar. 2020), p. 1015.

[31] A.H.S. Solberg et al. “Automatic detection of oil spills in ERS SAR images”. In: *IEEE Transactions on Geoscience and Remote Sensing* 37.4 (1999), pp. 1916–1924.

[32] M. Kubát, R. Holte, and S. Matwin. “Machine Learning for the Detection of Oil Spills in Satellite Radar Images”. In: *Machine Learning* 30 (2004), pp. 195–215.

[33] Camilla Brekke and Anne H.S. Solberg. “Oil spill detection by satellite remote sensing”. In: *Remote Sensing of Environment* 95.1 (2005), pp. 1–13.

[34] Diego Cantorna et al. “Oil spill segmentation in SAR images using convolutional neural networks. A comparative analysis with clustering and logistic regression algorithms”. In: *Applied Soft Computing* 84 (2019), p. 105716.

[35] Amir Adler et al. “A Deep Learning Approach to Block-based Compressed Sensing of Images”. In: *CoRR* abs/1606.01519 (2016). arXiv: 1606.01519.

[36] Byeongyong Ahn and Nam Ik Cho. “Block-Matching Convolutional Neural Network for Image Denoising”. In: *CoRR* abs/1704.00524 (2017). arXiv: 1704.00524.

[37] Haoliang Li, Shiqi Wang, and AlexC Kot. “Image Recapture Detection with Convolutional and Recurrent Neural Networks”. In: *Electronic Imaging* 2017 (Jan. 2017), pp. 87–91.

[38] Danial Maleki et al. “BlockCNN: A Deep Network for Artifact Removal and Image Compression”. In: *CoRR* abs/1805.11091 (2018). arXiv: 1805.11091.

[39] Jonathan Quijas and Olac Fuentes. “Removing JPEG blocking artifacts using machine learning”. In: *2014 Southwest Symposium on Image Analysis and Interpretation*. 2014, pp. 77–80.

[40] Haotong Qin et al. “Binary Neural Networks: A Survey”. In: *CoRR* abs/2004.03333 (2020). arXiv: 2004.03333.

[41] Matthieu Courbariaux and Yoshua Bengio. “BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1”. In: *CoRR* abs/1602.02830 (2016). arXiv: 1602.02830.

[42] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. “Extreme learning machine: Theory and applications”. In: *Neurocomputing* 70.1 (2006). Neural Networks, pp. 489–501.

[43] Guang-Bin Huang et al. “Extreme Learning Machine for Regression and Multiclass Classification”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 42.2 (2012), pp. 513–529.

[44] Guang-Bin Huang. “What are Extreme Learning Machines? Filling the Gap Between Frank Rosenblatt’s Dream and John von Neumann’s Puzzle”. In: *Cognitive Computation* 7 (June 2015), pp. 263–278.

[45] Guang-Bin Huang. “An Insight into Extreme Learning Machines: Random Neurons, Random Features and Kernels”. In: *Cognitive Computation* 6 (Sept. 2014), pp. 376–390.

[46] *Keyshot manual*. 2021. URL: https://manual.keyshot.com/manual/render-4/render-options/.

[47] *Matlab - Image Processing Toolbox*. 2021. URL: https://it.mathworks.com/products/image.html.

[48] Matlab. URL: https://it.mathworks.com/help/matlab/ref/rgb2gray.html.

[49] Nobuyuki Otsu. “A Threshold Selection Method from Gray-Level Histograms”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66.

[50] Connor Shorten and Taghi Khoshgoftaar. “A survey on Image Data Augmentation for Deep Learning”. In: *Journal of Big Data* 6 (July 2019). DOI: 10.1186/s40537-019-0197-0.

[51] *Keras - Image data preprocessing*. 2021. URL: https://keras.io/api/preprocessing/image/.

[52] Shreyas Fadnavis. “Image Interpolation Techniques in Digital Image Processing: An Overview”. In: *International Journal Of Engineering Research and Application* 4 (Nov. 2014), pp. 2248–962270.

[53] G. Wolberg. “Image morphing: a survey”. In: *The Visual Computer* 14 (1998), pp. 360–372. DOI: https://doi.org/10.1007/s003710050148.

[54] Aksh Patel. “Image Morphing Algorithm: A Survey”. In: 2015.

[55] volotat. URL: https://github.com/volotat/DiffMorph.

[56] Laurens van der Maaten and Geoffrey Hinton. “Visualizing Data using t-SNE”. In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605.

[57] Makoto Matsumoto and Takuji Nishimura. “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator”. In: *ACM Trans. Model. Comput. Simul.* 8.1 (Jan. 1998), pp. 3–30.

[58] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Tech. rep. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, Sept. 2014.

[59] Thibaud Ehret et al. "How to Reduce Anomaly Detection in Images to Anomaly Detection in Noise". In: *Image Processing On Line* 9 (2019), pp. 391–412.

[60] STMicroelectronics. URL: `https://www.st.com/en/embedded-software/x-cube-ai.html`.

[61] URL: `https://www.keil.com/pack/doc/CMSIS/Core/html/group__intrinsic__SIMD__gr.html`.

[62] arm. URL: `https://github.com/ARM-software/CMSIS_5`.