Ecosystem for COllaborative Manufacturing PrOceSses – Intra- and Interfactory Integration and AutomaTION
(Grant Agreement No 723145)

D5.4 Continuous Deep Learning Toolkit for Real Time Adaptation II

Date: 2019-02-27

Version 1.0

Published by the COMPOSITION Consortium

Dissemination Level: Public

Co-funded by the European Union's Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 723145

Document control page

Document file: D5.4 Continuous deep learning toolkit for real time adaptation II v1.0.docx
Document version: 1.0
Document owner: ISMB

Work package: WP5 – Key Enabling Technologies for Intra- and Interfactory Interoperability and Data Analysis
Task: T5.2 – Continuous Deep Learning Toolkit for real time adaptation
Deliverable type: O

Document status: ☒ Approved by the document owner for internal review
☒ Approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Paolo Vergori (LINKS)	29-11-2018	Definition of table of contents
0.8	Paolo Vergori, Nadir Raimondo, Luigi Giugliano (LINKS)	19-02-2019	Updated according to the new development
0.9	Luigi Giugliano (LINKS)	25-02-2019	Merged the review of Peter Haigh (TNI-UCC)
1.0	Luigi Giugliano (LINKS)	27-02-2019	Merged the review of Nacho González (ATOS)

Internal review history:

Reviewed by	Date	Summary of comments
Peter Haigh (TNI-UCC)	23-02-2019	Approved with comments
Nacho González (ATOS)	27-02-2019	Approved with minor comments

1 Executive Summary

The present document named “D5.4 Continuous deep learning toolkit for real time adaptation II.docx” is a public deliverable of the COMPOSITION project, co-funded by the European Union’s Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 723145. It reports the final results of task “5.2 – Continuous Deep Learning Toolkit for real time adaptation” that foresees its development in work package 5 “Key Enabling Technologies for Intra- and Interfactory Interoperability and Data”.

The document owner is ISMB and in this final version is a reiteration of the submitted at M16, namely D5.3 Continuous deep learning toolkit for real time adaptation I. This version highlights the final results after a second iteration of task 5.2, regarding the development of a Deep Learning Toolkit for real time adaptation. A comprehensive data assessment is provided alongside prove of concept results with an in-depth theoretical validation, for each of the addressed project’s use cases. Although this version has to be considered as final, there may still be a few software modifications since this deliverable is submitted at M30 whereas the project ends at M36.

For the sake of completeness, this document adopts a conservative approach in which valuable information from the previous iteration of this deliverable has been maintained in the annexes. Following this approach, the result is a solid document that can be redistributed among partners without lacking cross references and missing the opportunity to follow the logical evolution of tests, trials and deployments phases.

2 Abbreviations and acronyms

Table 1: Abbreviation and acronyms table

Acronym	Description
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application programming interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DoW	Document of Work
DLT	Deep Learning Toolkit
GPU	Graphical Processing Unit
HDFS	Hadoop Distributed File System
LSTM	Long-Short Term Memory
MAE	Mean Absolute Error
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSE	Mean Squared Error
NN	Neural Network
OGC	Open Geospatial Consortium
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
RMSE	Root-Mean Squared error
RMSLE	Root-Mean Squared Logarithmic Error
RNN	Recurrent Neural Network

3 Introduction

3.1 Summary

This document describes the development carried out in COMPOSITION's project task 5.2, named Continuous Learning Toolkit for Real Time Adaptation.

The document is structured in nine sections and after a comprehensive analysis of the state-of-the-art of relevant fields, a comparison among existing frameworks of interest is presented. The COMPOSITION project's use cases, in which this task has been involved, have been evaluated and the end users' historical datasets have been assessed through a qualitative based validation process. The main chapter (7) is the one relative to the analysis of the Deep Learning Toolkit component that was developed within T5.2. The developed component is going to be deployed in the aforementioned use cases, in which all project's end users are going to be involved. In this chapter, results will be presented alongside the used methodology. Specifically, results on extensive tests with real data about the two different use cases. Following this chapter there will be a detailed explanation on the deployment procedure of our component. Last but not least, the section regarding conclusion and future work will be provided.

It is worth mentioning that the expected TRL of the component developed in task 5.2 is four, according to the DoA.

3.2 Background

As the challenges of Industry 4.0 are absorbed by the research world, bringing together world-class manufactures and the academic world. Science fiction movies has drawn for decades a dystopian research reality where machines take over men's labour and even worse. The COMPOSITION project treats AI as a powerful resource and tackles real world problems, such as predictive maintenance and raw material market price estimations by advancing the state-of-the-art of current technologies.

As new algorithms are being developed, their time efficiency and resource consumption is progressively decreasing. The number of possible applications in which AI is applicable is becoming almost endless. It is clear to the scientific community that the real power of Artificial Neural Networks (ANNs) will not be achieved by withholding intellectual proprieties over algorithms or frameworks, but by open source licences that are progressively improved by the scientific community. However the true power over predictions' accuracy dwells in the data ownership.

Regarding the tool's development described in this document, it is worth mentioning that the aim is not to create a Swiss knife tool for every application, but a tailored solution that fits a complex ecosystem from its roots to its leaves. After outlining these scenarios, it is easy to understand why the Deep Learning Toolkit (DLT) described in this document will have as many declinations as the use cases in which it will be deployed. Each solution will be specifically developed for the actions that will be required to take and will be based on the available historical data. As will be made clear at the end of this reading, the success rate and the convergence period will be drastically dependent by the amount of data available in each of the scenarios.

The first iteration of this deliverable was focused on data analysis and assessment from historical datasets and on synthetic data demonstration of the potential of ANNs. In the end, a first deployment of a trained ANN that uses real world data is also described in its deployment in the lab-scale testing environment.

The second iteration and the final one are likewise focused respectively on data analysis and assessment of historical datasets, but with a deeper approach on two use cases. Together with the analysis, the results of different ANNs trained on real data (historical and live) coming from our partners will be provided.

NB the previous analysis on synthetic data have been moved to the Annex I for simplify the structure and the reading of this document.

4 State-Of-The-Art analysis

4.1 Intro to machine learning

Machine Learning (ML) is the branch of computer science concerned with the development of algorithms and techniques allowing computers to learn from experience/data.

ML arises at the intersection of several research fields:

- Artificial intelligence: smart algorithms to successfully interact with the environment.
- Statistics: inference from samples.
- Data mining: search through large volumes of data.
- Computer science: efficient algorithm and complex models.
- Pattern recognition: analyse and interpret data looking for recurrent structures.

A well-known and widely accepted ML definition, due to (Mitchell, 1997) and dating back to 1997 states that:

«A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E»

The ML approach is conceptually structured in two independent stages:

- Training: analyse input dataset → gain understanding → fit a model that explains the data.
- Deploy: use the model to make prediction of new data.

Figure 1 illustrates both Mitchell's definition and the two stages in the real case example. The task T is to predict the content of a picture. The experience E is the input dataset composed of a list of pictures and the associated expected labels. The computer program is the model trained with the dataset in the first stage: it is a trial and error process in which the model is used to predict the content of pictures. The prediction is compared to the target label computing an error metric (the performance measure P) and finally the error is used to perform a finer tuning of the model. Repeating this process iteratively progressively improves the model performance until it can be deployed and applied to predict the content of new unseen and unlabelled images.

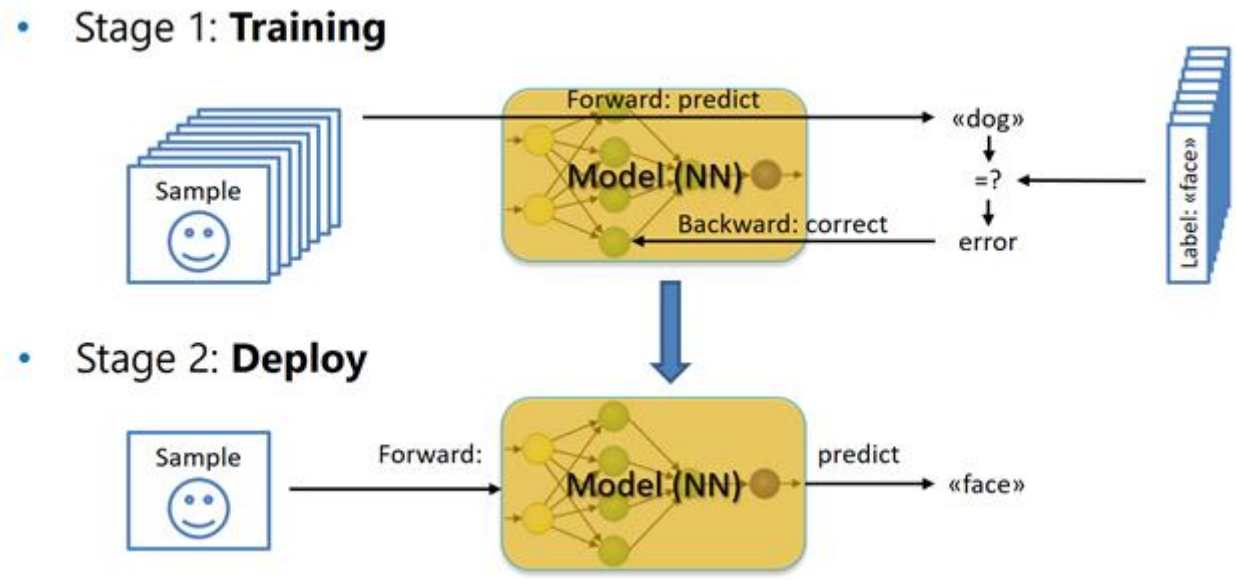

Figure 1: example of machine learning stages for the task of image content prediction

ML is therefore a good approach in a plethora of different situations and it is preferable to the traditional computer science paradigm of sequential and object-oriented programming where each problem require a custom application.

4.2 Intro to neural network and deep learning

Artificial Neural Networks are defined in (Wikipedia, n.d.) as a computing system inspired by the biological neural networks that constitute mammalian cerebral cortex. Such systems learn (progressively improve performance on) tasks by considering examples, generally without task-specific programming.

An ANN is based on a collection of connected units or nodes called artificial neurons (analogous to biological neurons in an animal brain). Each connection (synapse) between neurons can transmit a signal from one to another. The receiving (postsynaptic) neuron can process the signal(s) and then signal neurons connected to it. In common ANN implementations, the synapse signal is a real number and the output of each neuron is calculated by a non-linear function of the sum of its inputs. Neurons and synapses typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal it sends across the synapse.

Neural Network models normally have a hierarchical organization into distinct layers of neurons. Each layer of neurons represents a level in that hierarchy. The number of layers supported by a model is theoretically non-limited but it was generally restricted to few layers. The limitations were overcome in recent algorithms evolution that rely on:

1. The increased availability of data.
2. The enhancement of computing resources.

Neural networks with multiple hidden layers are referred to as Deep Neural Networks.

4.3 Deep Feed Forward NN

A feed forward neural network is a collection of neurons connected in an acyclic graph. The data travel forward, from the first (input), to the hidden nodes (if any) and finally through the output nodes. In other words, the outputs of the neurons of a layer can only become inputs of a deeper layer.

This network type came from Frank Rosenblatt's machine-learning algorithm named "Perceptron" (Rosenblatt, 1958). Two kinds of perceptron have been generated as described in (Wikipedia, n.d.) (Frank, 1962):

- Single-layer perceptron network: single layer of output nodes; the inputs are fed directly to the outputs via a series of weights. In this way, it can be considered the simplest kind of feed-forward network. Single-unit perceptron's are only capable of learning linearly separable patterns.
- Multi-layer perceptron networks or feed forward neural networks: multiple layers of interconnected computational unit. Each neuron in one layer has directed connections to the neurons of the subsequent layer.

Figure 2 shows an example of 3-layer Feed Forward Multi-layer perceptron with two inputs, two hidden layers with 3 neurons each and one output layer. All the connections are between adjacent layers and neurons within a single layer share no connections.

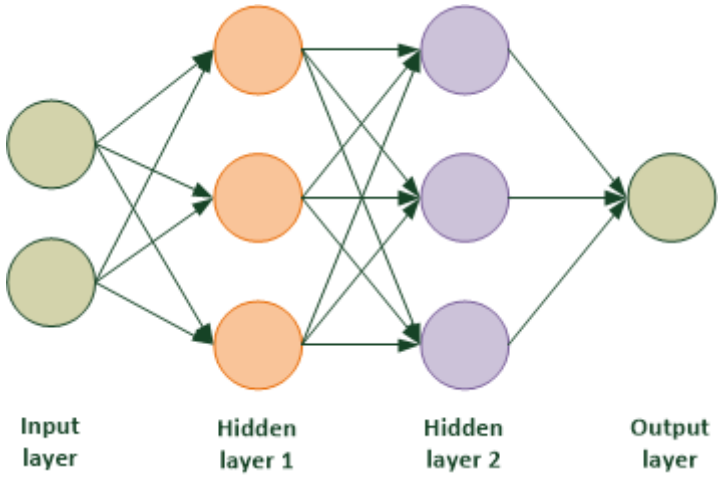

Figure 2: network layers

4.4 Recurrent neural network and long-short term memory network

The excerpt from literature continues in (Wikipedia, n.d.), where all this information is widely available. A brief and concise dissertation continues in the following.

A Recurrent Neural Network (RNN), as defined in (Wikipedia, n.d.), is a class of artificial neural networks where connections between units form a directed cycle. This allows it to exhibit dynamic temporal behaviour. Unlike feed forward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs.

The back-propagation of information, using gradient descent, is used to move the network's weights to better guesses. As described in (Wikipedia, n.d.), artificial neural networks with gradient-based learning methods and back propagation can suffer some difficulties in training due to the vanishing gradient problem. In such methods, each of the neural network's weights receives an update proportional to the gradient of the error function with respect to the current weight in each iteration of training. The problem is that, in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.

Long-Short Term Memory (LSTM) network (Wikipedia, n.d.) was introduced by Hochreiter and Schmidhuber in 1997 (Hochreiter & Schmidhuber, 1997) to overcome the vanishing gradient problem of traditional RNNs. An LSTM network contains a (memory) cell. An LSTM cell "remembers" a value for either long or short time periods. The key to this ability is that it uses the identity or no activation function within its recurrent connection. In other words, the remembered value of the cell is not iteratively modified because there is the identity or no activation function through which the value flows. This is the key for the gradient not to tend to vanish when an LSTM network is trained with back propagation through time.

A "standard" LSTM block contains three gates that control or regulate information flow: an input gate, an output gate and a forget gate. These gates compute an activation (usually using the logistic function). These gates can be thought as conventional artificial neurons. Thus, each of the gates has its own parameters (i.e. weights and biases from possibly other units outside the LSTM block). Their output is multiplied with the output of the cell or the input to the LSTM to partially allow or deny information to flow into or out of the memory. More specifically, the input gate controls the extent to which a new value flows into the memory, the forget gate controls the extent to which a value remains in memory and the output gate controls the extent to which the value in memory is used to compute the output activation of the LSTM block.

5 Frameworks comparison

5.1 Introduction to deep learning frameworks

Even well-known machine learning algorithms are quite complex to implement from scratch: it's possible to get lost in maths details and computational technicalities. To address this issue, in recent years a plethora of software frameworks dedicated to machine learning and deep learning emerged, providing simplified API and efficient code reuse. Some of them come from academia, while others are released from global companies such as Google and Microsoft. Each one has specific features, pros and cons: there is no such a thing as the absolute best framework. As a result, a convenient framework choice has to be taken based on the specific requirements of the end task to address.

Hands-on comparisons and benchmarks of so many heterogeneous frameworks would require a lot of effort to set up different development environments each with its own API. Hence, a feature-based review has been carried out to narrow down the number of candidates.

As part of task 5.2, an extensive review has been conducted of the most recent and popular frameworks supporting deep learning algorithms. The results are presented in the following sections.

The considered candidates include:

- Caffe
- CNTK
- Deeplearning4j
- TensorFlow
- Theano
- Torch
- H2O.ai
- Wolfram Mathematica
- Neural Designer
- Apache Singa
- Chainer
- OpenNN
- MXNet

5.2 Evaluation criteria

Several different criteria have to be considered to evaluate frameworks for machine learning. The most relevant are listed below.

- Date of first and last release: older frameworks have better chances to be mature, consolidated, stable and support a wider number of algorithms. On the other hand, recent frameworks have good chances to focus on state-of-the-art algorithms and have more modern overall architecture. In any case, actively developed frameworks must be preferred.
- Usage licence: In COMPOSITION, there is interest to the open/closed source dichotomy as well as to the possibility of embedding in commercial products with a view to commercial exploitation.
- Documentation, adoption and commercial support: the availability of detailed and up to date documentation is an important consideration for ML framework adoption as for any other piece of software. The same applies to the end user, where the the availability of tutorials, examples and blog posts are important.
- Supported hardware: while the model training based on historical data can be carried out offline with workstations providing the necessary computational power (usually Intel x64 CPUs, with the optional

support of GPUs), once deployed the model has to be able to deliver prediction and perform continuous learning. These activities are less computationally intensive than the initial training and can be carried out on cheaper hardware: such as SoC, embedded or mobile devices, which may use ARM CPUs and offer no support to GPU computing. This is possible only if the framework supports the target hardware platform, therefore it is desirable to adopt a framework with the widest possible hardware support.

- Supported platform: the same considerations related to supported hardware applies to software platforms: in order to allow deployment of a wide range of different devices, the adopted framework has to be multiplatform.
- Algorithm supports: frameworks offer different support to various families of machine learning algorithms. This is a multifaceted issue as the implementation can be not just either present or missing, but partial as well as sub-optimal under many different aspects. Another related feature is the availability of pre-trained models (not just the algorithm itself, but also a trained version of it to tackle specific tasks); this allows to work in an incremental way, building on top of previous successful models with efficient reuse of training effort.
- Core Language: the language in which the core of the framework is implemented. Typically affects training speed as both low-level languages and languages targeted to numerical computation are faster to execute that high level, general-purpose languages.
- API language: frameworks usually offer APIs in different languages for convenience of usage. High-level languages such as python are faster to write and easier to read compared to C/C++ so that such APIs bring a consistent speedup in model implementation. The higher number of supported languages, the better.
- Auto-differentiation support: most neural networks rely on variations of Gradient Descent algorithm to perform training. In particular, the gradient of a cost function has to be evaluated many times. This can be done either analytically or numerically. In the first case the gradient function has to be known and provided to the framework by the programmer, while in the second case the framework is able to numerically approximate with no additional knowledge. The latter is less precise and slightly slower but applies to much more wide range of real problems.
- API abstraction level: the frameworks can provide APIs at different abstraction levels. The lower the level, the greatest the control over the algorithmic details. The higher the level, the shorter and more readable the code, resulting in faster test and comparison of different models.
- Parallel/GPU computing support: given that the training of Deep Neural Network over large datasets is computationally large, it is possible to reduce the execution time leveraging multithreading: the computation workload is delegated to multicore hardware such as GPUs where it can be massively parallelized.
- Distributed architectures support: another way to approach computational workload of training is to split it across a cluster of machines exploiting the network infrastructure. This approach can be combined with GPU computing for even higher speedup.

5.3 Comparison

In this section, the comparison of frameworks is detailed. Only actively developed frameworks were considered. Furthermore, being interested in deep neural network state-of-the-art, a bigger attention has been devoted to neural network capabilities and in particular to Feed Forward NN and Recursive NN. Convolutional NN and Auto encoders, despite being innovative algorithms, are less relevant for the COMPOSITION use cases; therefore, they are not explicitly considered in the comparison tables.

The original assessment was created in October 2016 (M2) but the data reported in this document has been updated at September 2017 (M13).

Table 2 to Table 6 compares the frameworks under different features:

Table 2: comparison of frameworks - creator, first and last releases

Framework	Creator	Year first release	Year last release
Apache Singa	Apache Incubator	2015	2017
Caffe	Berkeley Vision and Learning Center	2013	2017
Caffe2	Facebook	2017	2017
Chainer	PFI/PFN	2015	2017
Deeplearning4j	Skymind engineering team	2014	2017
H2O	H2O.ai	2011	2017
Microsoft Cognitive Toolkit	Microsoft Research	2016	2017
MXNet	Distributed (Deep) Machine Learning Community	2015	2017
Neural Designer	Artelnics	2014	2017
OpenNN	Artelnics	2003	2017
TensorFlow	Google Brain team	2015	2017
Theano	Université de Montréal	2008	2017
Torch	Ronan Collobert, Koray Kavukcuoglu, Clement Farabet	2002	2017
Wolfram Mathematica	Wolfram Research	1988	2017

Table 3: comparison of frameworks – licensing

Framework	Licence	Private & Commercial use	Open source
Apache Singa	Apache 2.0	Yes	Yes
Caffe	BSD 2-Clause	Yes	Yes
Caffe2	BSD 2-Clause	Yes	Yes
Chainer	MIT	Yes	Yes
Deeplearning4j	Apache 2.0	Yes	Yes
H2O	Apache 2.0	Yes	Yes
Microsof Cognitive Toolkit	MIT	Yes	Yes
MXNet	Apache 2.0	Yes	Yes
Neural Designer	Proprietary	---	No
OpenNN	GNU LGPL	Yes	Yes

TensorFlow	Apache 2.0	Yes	Yes
Theano	BSD 3-Clause	Yes	Yes
Torch	BSD License	Yes	Yes
Wolfram Mathematica	Proprietary	---	No

Table 4: comparison of frameworks - supported platforms & languages

Framework	Platform	Core Language	API Language
Apache Singa	Linux, Mac OS X	C++, Python	Python, C++
Caffe	Windows, Linux, OS X	C++, Python	C++, command line, Python, Matlab
Caffe2	Windows, Linux, OS X	C++, Python	Python, Matlab
Chainer	Linux	Python	Python
Deeplearning4j	Windows, Linux, OS X, Android	C, C++	Java, Scala, Clojure, Python (via Keras)
H2O	Windows, Linux, OS X	Java	Java, Python, R, Scala
Microsof Cognitive Toolkit	Windows, Linux	C++	Python, C++, Command line, BrainScript
MXNet	Windows, Linux, OS X	C++, Python, Julia, Matlab, Go, R, Scala	C++, Python, Julia, Matlab, JavaScript, Go, R, Scala, Perl, Wolfram Language
Neural Designer	Windows, OS X, Linux	C++	Graphical user interface
OpenNN	Windows, Linux, OS X	C++	C++
TensorFlow	Windows, Linux, OS X	C++, Python	Python, (C/C++public API only for executing graphs)
Theano	Windows, OS X, Linux	Python	Python
Torch	Linux, Android, OS X, iOS	C, Lua	Lua, LuaJIT
Wolfram Mathematica	Windows, OS X, Linux	C++	Command line, Java, C++

Table 5: comparison of frameworks - GPU & distributed computing

Framework	Parallel/GPU Computing	Distributed Architecture
Apache Singa	Yes (CUDA)	Yes
Caffe	Yes (CUDA, OpenMP)	Yes
Caffe2	Yes (CUDA, OpenMP)	Yes
Chainer	Yes (CUDA)	Yes (with ChainerMN)
Deeplearning4j	Yes (CUDA, OpenMP)	Yes (Apache Spark)

H2O	Not directly	Yes (Apache HDFS, Apache Spark; Cloud: Amazon EC2, Google Compute Engine, and Microsoft Azure)
Microsof Cognitive Toolkit	Yes (CUDA, OpenMP)	Yes
MXNet	Yes (CUDA, OpenMP)	Yes
Neural Designer	Yes (CUDA, OpenMP)	Yes (Amazon WS)
OpenNN	Yes (CUDA, OpenMP)	No
TensorFlow	Yes (CUDA)	Yes
Theano	Yes (CUDA, OpenMP)	Partial
Torch	Yes (CUDA, OpenMP)	Partial
Wolfram Mathematica	Yes (CUDA, OpenMP, OpenCL)	Yes

Table 6: comparison of frameworks - algorithm support

Framework	Auto Differentiation	Pre-trained models	RNN	CNN
Apache Singa	Yes	Yes	Yes	Yes
Caffe	Yes	Yes	Yes	Yes
Caffe2	Yes	Yes	Yes	Yes
Chainer	Yes	Through Caffe's model zoo	Yes	Yes
Deeplearning4j	Computational Graph	Yes	Yes	Yes
H2O	Yes	No	Not directly	Not directly
Microsof Cognitive Toolkit	Yes	No	Yes	Yes
MXNet	Yes	Yes	Yes	Yes
Neural Designer	Yes	?	No	No
OpenNN	Yes	No	No	No
TensorFlow	Yes	Yes	Yes	Yes
Theano	Yes	Through Lasagne's model zoo	Yes	Yes
Torch	Through Twitter's Autograd	Yes	Yes	Yes
Wolfram Mathematica	Yes	Yes	No	Yes

In order to compare the popularity and the level of development activity around each framework, several charts are reported.

Figure 3 compares the trends of number of questions asked about frameworks on the popular platform Stack Overflow along several years. Only few frameworks are considered.

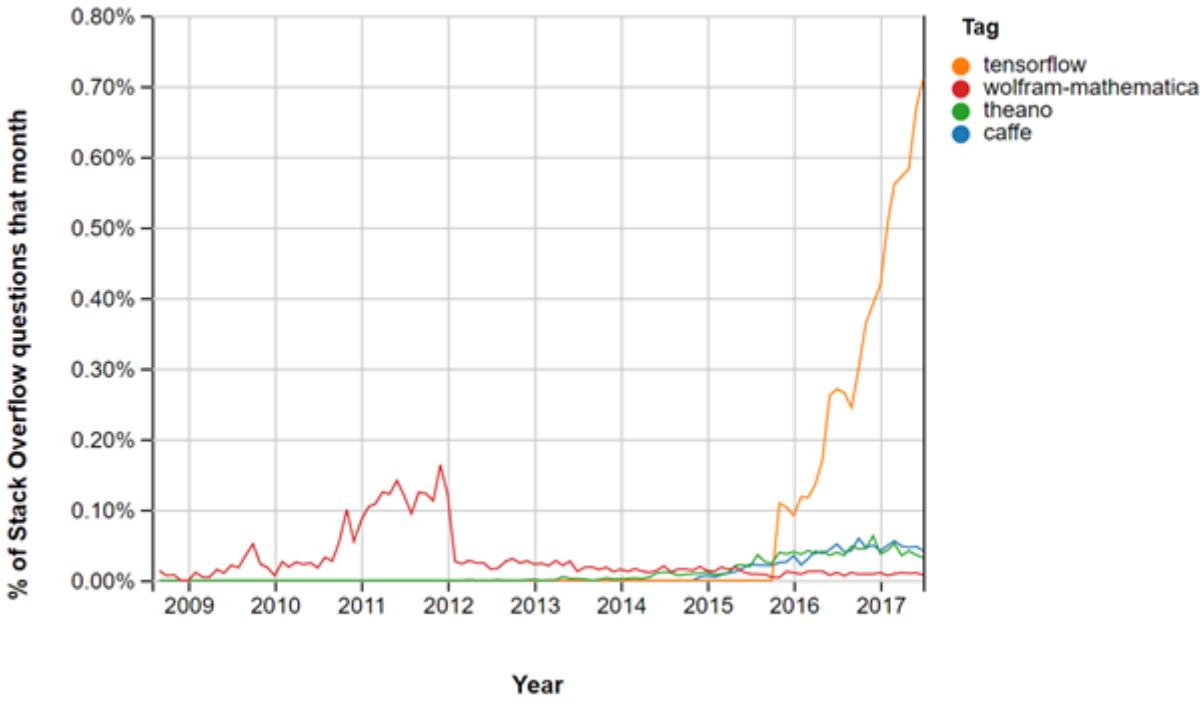

Figure 3: trends of Stack Overflow questions

A more comprehensive popularity comparison was obtained through the statistics of GitHub.com where all open source frameworks have their public repository. Weekly number of commits along a one-year period (09/2016 to 09/2017) in Figure 4 gives a detailed insight about recent development trends, while the average values reported in Figure 5 let the most active frameworks emerge.

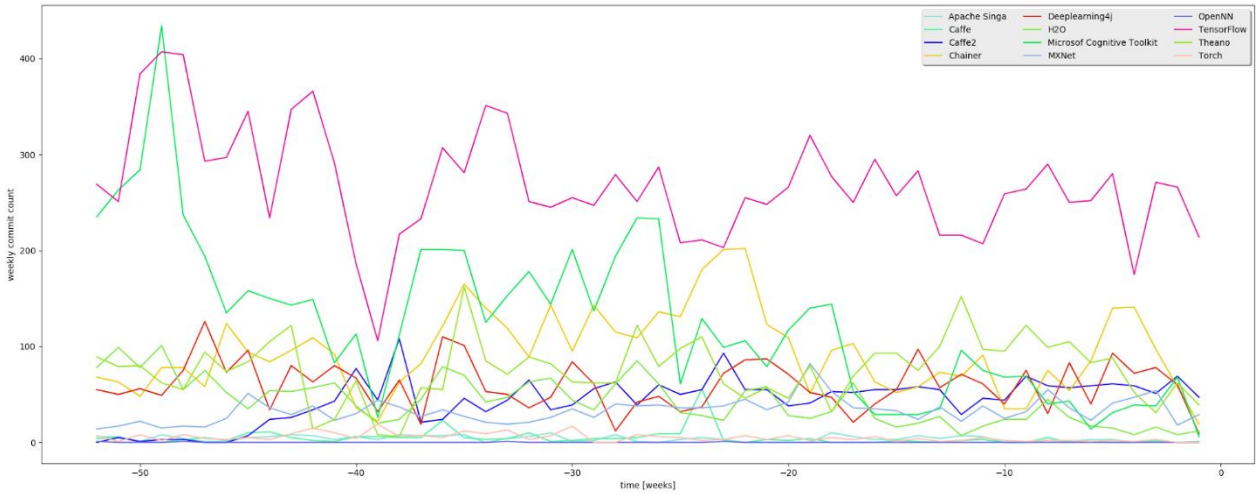

Figure 4: trends of weekly GitHub commits across the last year

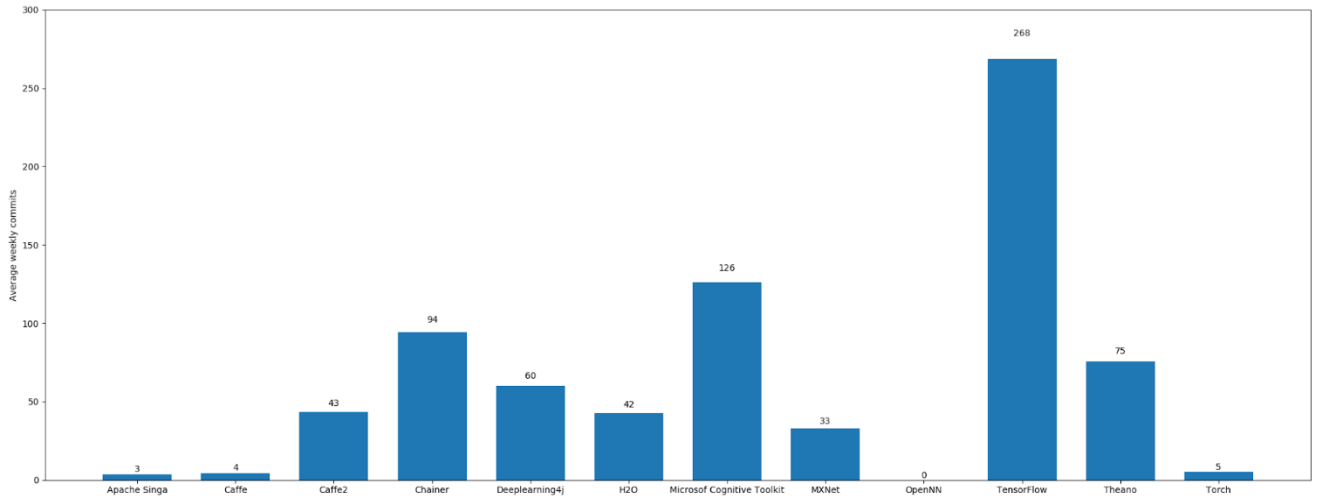

Figure 5: average numbers of weekly GitHub commits in the last year

Another widely adopted popularity metric is the trend of a topic in Google search. In Figure 6 the frameworks are compared in terms of weekly search interest (normalized to the range [0-100]) since 2013.

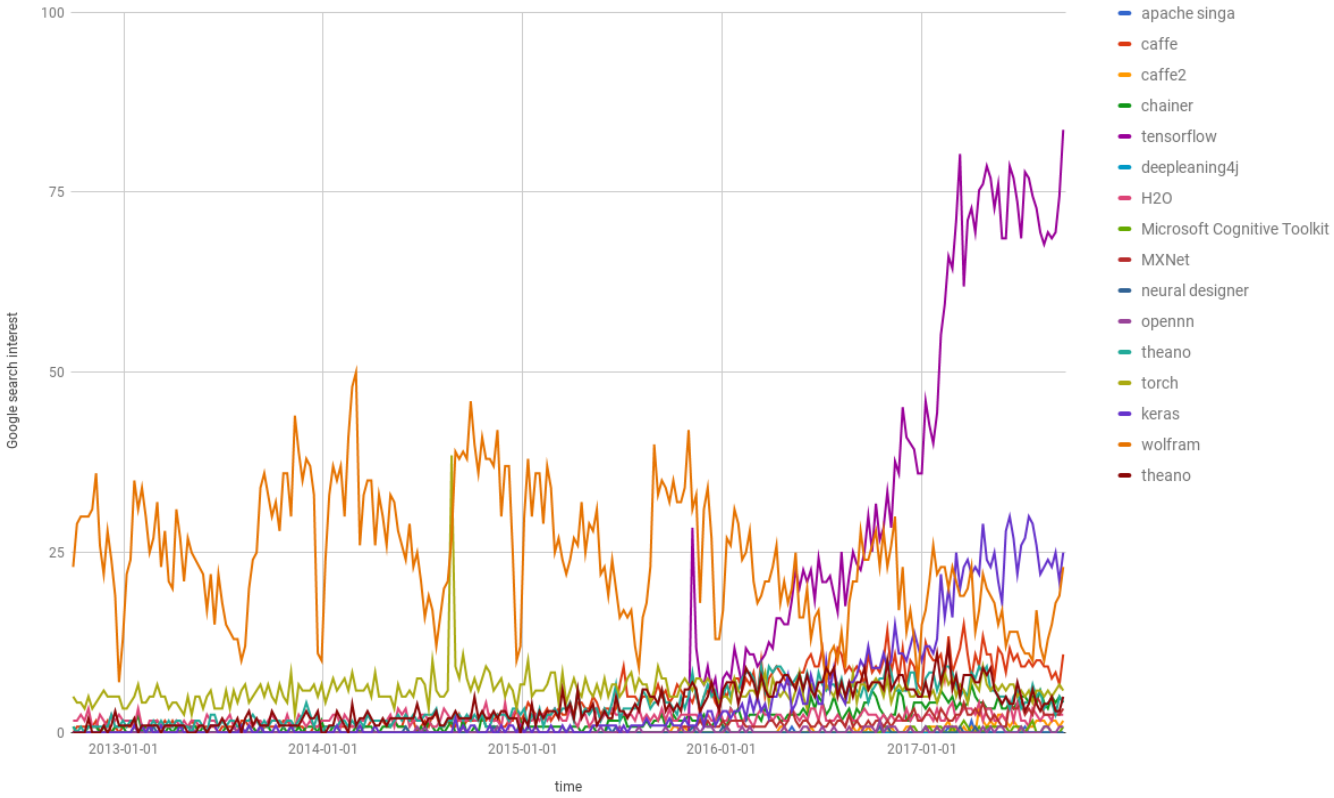

Figure 6: Google search trends

Some considerations emerged from the data reported so far.

The majority of deep learning frameworks are open source and can be used free of charge even for commercial purposes (as long as the original code is not modified). On the other hand, the most famous commercial framework: Wolfram Mathematica (and more in general the whole ecosystem of Wolfram products) is up to date and full featured but the licence pricing options, ranging from 3000 to over 9000 euros per license, discourages the adoption since there are valid free alternatives.

In particular, open source frameworks are developed, maintained and used by top universities machine learning research groups, software foundation's/communities and more and more frequently, from global tech

companies such as Google and Facebook. The resulting frameworks are the ones adopted both for research purposes and for the implementation and deployment of industrial-grade commercial products with global distribution.

Because of these reasons, an open source framework has been preferred for the development of the COMPOSITION Deep Learning Toolkit. During the drafting of this assessment in October 2016, the number of candidates has been reduced to a restricted pool based on the features and popularity at that date.

Additional advantages and disadvantages are presented in Table 6 to support the final framework choice. It is worth noting that framework evolution is very fast and new features are incorporated every few months, so the trade-off of pros and cons of each solution are likely to be changed by the time of writing.

Table 7: pros and cons of open source frameworks

Framework	Advantages	Disadvantages	Notes
Caffe	Many extensions C++ --> multiplatform. Lots of pretrained model on its ModelZoo site.	Most command line interface. Need to write CUDA for GPU layers. Not good for recurrent networks. Not extensible, may be difficult to apply outside Computer Vision.	Most popular for Computer Vision tasks and CNN.
H_2O	Fast and scalable. Very well documented API. High level API --> easy to use. Very convenient Grid Search across hyperparameters space.	Limited choice of ANN models: no RNN and CNN (but can handle arbitrarily complex model by using other frameworks as core).	Other companion products allow extending H_2O functionalities for GPU computing and for scaling on Apache HDFS & Spark. In addition, it is possible to deploy on most famous commercial cloud services.
Microsoft Cognitive Toolkit	C++ --> multiplatform (but not working on ARM). Good RNN implementation.	Not yet usable for a variety of tasks. Most command line interface.	In the beginning, mainly adopted for speech recognition and natural language processing (NLP) tasks.
TensorFlow	Python and C++ interface C++ --> multiplatform. Faster than Theano. Industrial grade deployment system. Most popular framework, extremely actively developed.	RNN are still suboptimal. Bidirectional RNN not yet available. Slower than other frameworks and fatter than Torch. Few pre-trained models.	Meant as a replacement of Theano.
Theano	Has implementation of most SoA networks directly or as higher-level framework. Python interface.	Deployment require python interpreter --> overhead (less attractive for industrial use). Untidy legacy architecture. Steep learning curve for low-level Theano API. Long compile time (fatter than Torch).	First learning framework, mainly used in academic On-top framework: Keras, Lasagne, Blocks.

Torch	Lots of modular pieces easy to combine, and pretrained models. Excellent for convolutional network (better than TensorFlow or Theano). Good for RNN through an extension. More flexible than TF/Th: no graph -> better for beam search. Lua is fast.	Lua is not mainstream language. Maybe difficult to integrate with other software components. Need to write code for training. Spotty documentation.	Very popular for Computer Vision tasks and CNN.

As for the final choice of frameworks to adopt in COMPOSITION:

- Theano was discarded since TensorFlow is more advanced
- Torch and Caffe were discarded since more focused on Computer Vision tasks and more suitable to academic research purposes than deployment in production contexts.
- Microsoft Cognitive Toolkit was discarded because at the time had limited functionalities and was mainly focused on NPL tasks.

In the end, TensorFlow was chosen as the main development environment, due to its growing popularity, its APIs operating at different abstraction levels (allowing to trade-off between ease of coding and control of algorithm details). Being developed and adopted by Google for its AI projects seemed to offer significant advantages in relation to community width, continuous development, quality of the documentation and efficiency of deployment systems.

In addition, H2O, were included for fast testing of Feed Forward Deep Neural Network do to its training speed, ease of use, and to the powerful built-in Grid Search functionality.

As the project progressed the need for Recurrent Neural Networks emerged, this is discussed further in sections 6 and 7 of this document. A new evaluation was undertaken to include RNN support as a requirement. The outcome of this evaluation was that although other candidates such as Microsoft Cognitive Toolkit now offered good capability for the project, the incumbent toolkit TensorFlow had developed adding new features including RNN support. Changing the framework half way through the project would have proved a significant risk and so it was decided to remain with TensorFlow.

6 Inter and Intra-factory end users' historical data assessment

In this chapter, the results of the assessment conducted over the historical datasets provided by industrial partners are reported. The availability of such datasets is mandatory for a component like the Deep Learning Toolkit, whose core is mainly composed of deep artificial neural network models.

In fact, in order to achieve state-of-the-art prediction accuracy, artificial neural networks need to be extensively trained over large datasets. There is a linear dependency between the model complexity and the amount of data required: the deeper and more complex the model, the larger the training set is required. The Deep Learning Toolkit is going to be deployed in an already trained form, based on the addressed scenarios. Once deployed, it will process live data streams provided by the Big Data Analytics component, producing meaningful predictions and updating the model whenever enough information is processed and a new one is available. In order to adapt to future variations of patterns and trends, the deep learning toolkit bases its implementation on continuous learning, refining its training by analysing small batches of live data.

In order to be used in a supervised machine learning framework, each historical dataset has to be organized as a list of samples. Each sample is made of two parts: a vector of features, named X (e.g. values sampled from different sensors at the same time) and a corresponding scalar target value Y. The number of features and their type (int, float, string) can be various, but it must be fixed for all the samples of a dataset: consistency is mandatory and any kind of heterogeneity within a dataset is not allowed. The scalar vector Y also demands consistency and represents a target that is compulsory for each sample. Below a graphical representation of X and Y:

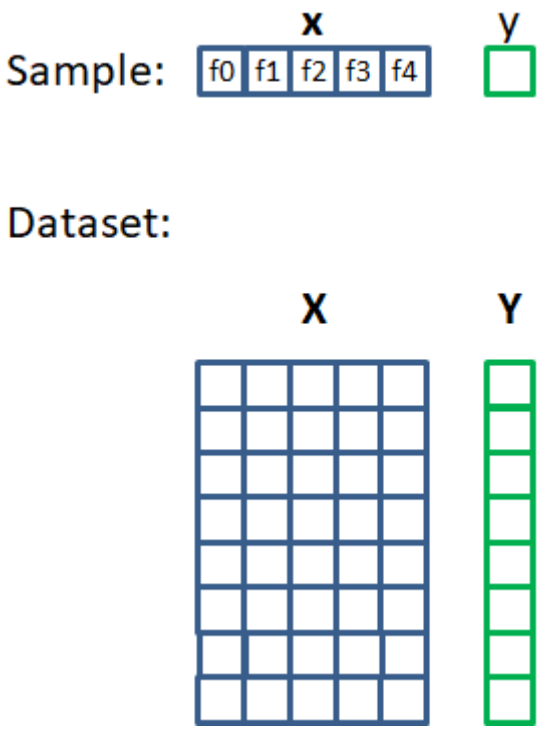

Figure 7: dataset structure

The rest of this chapter is structured per use case: for each use case in which the deep learning toolkit is involved and therefore one or more artificial neural network is going to be deployed, an in-depth analysis is performed. The following sections then discussed whether the provided data are adequate for the deep learning toolkit to perform its training actions, and if not, which aspects of the datasets are unfit, highlighting possible solutions to tackle the problem.

The references made in this document refer to the updated use cases' list, as defined in the most recent version of deliverable D2.1. In the following section the assumptions made are highlighted:

- Only the use cases in which the DLT is expected to contribute has been considered.

- All the use cases where the DLT will contribute are discussed (the DLT will not be part of other use cases).
- There is no assessment of non-tabular data such as pictures, maps, and textual descriptions.
- In the followings are linked the pages in which the corresponding data and use cases are assessed.

In the (hopefully) unlikely event of a historical dataset and therefore being impossible to perform the initial training, a model can be deployed untrained. It then would use only live data streams to perform both the training and the learning phases. In this case, a quite considerable amount of time must be considered as a transitional period that is required to reach the performance of an equivalent trained model and therefore being able to deliver any meaningful prediction.

Occasional missing values for some features for a small amount of samples can be dealt with, as long as they are minimal. Data series can be easily chunked and converted to a dataset format, but in order to be relevant for supervised learning they have to be sampled with a fixed frequency: sparse occasional samples are of limited or no use. As a rule of thumb, medium sized dataset counts 10K or more samples. Depending on the specific challenge the component will require to solve the problem, as the current state-of-the-art performance of any deep learning algorithm, 100K or more balanced samples are required for providing a more relevant training.

Particular relevance is required when considering the word balanced because it is the key in this topic: feeding models with millions of samples in normal state and hundreds of samples in fault state is not an option. In those cases, it is recommended to under sample the amount of data in the normal state, balancing the input for the model. Hence, when talking about the category of classification problems, like the ones we are referring to, it is possible to reduce the amount of data of most of the intra-factory use cases. If it is necessary to have 100K balanced samples, it is required to have $\frac{100K}{NumberOfClasses}$ for each class the model has to identify.

In the second iteration of COMPOSITION, some criticalities identified posed some major concerns with the applicability of the DLT ANN in some of the previously identified use-cases. In particular, after the mid-project updates, the DLT does not fit the following UCs' scenarios:

- Maintenance Decision Support (UC-KLE-1): the dataset does not fit the requirements needed to properly train a neural network. Due to the lacking of historical dataset on which train the network but moreover for the absence of faults or a pattern of faulty data

Data assessment previously carried out for these use cases can be found in annex I section 10.1.

6.1 Predictive maintenance (UC-BSL-2)

6.1.1 Background

The Deep Learning Toolkit component is expected to produce the latest prediction on the next expected failure of the oven blower machine, based on the continuous input of sensors data streams.

BSL provided a large dataset related to four reflow ovens (Brady, Tachy, Rhythmia and NMD).

During the first iteration of the project, Brady oven has been chosen for UC-BSL-2 intra-factory use-case due to the larger number of features available in its historical datasets (detailed in section 6.1.2). Accordingly to this choice, the initial study and experiments were carried on the Brady's dataset producing conclusions that are strictly bounded with the nature of its dataset.

On May 3, the partner Boston Scientific Limited (BSL) decided that Brady was not suited anymore for the intra-factory interoperability layer. Instead, Rhythmia was elected as the new oven that will be used for the COMPOSITION project. The reason of this "switch" can be address to the nature of the two ovens; Brady is a test oven while Rhythmia is a production oven.

By all means, this switch drops most of the previous conclusion that have been drawn from the study of Brady. The difference of the two datasets will be detailed in the section 6.1.2.1.

6.1.2 Data Overview

6.1.2.1 Legacy data

The term legacy data refers to all the data that was already present, logged and probably used before the start of this project. In this use case the legacy data is composed of the logs of sensors and events from the BSL' ovens.

For each BSL oven, the dataset encompasses data files, one per day, covering the period 2008-2017. Actually, the start point of the recording and the time span is different for each oven, varying between 2008 and 2013. Each file is structured as a list of records, one per row. Records are sampled every 5 minutes and contain, in addition to the timestamp, the logs of all the blowers inside machine. The number of blowers differs from reflow to reflow (e.g. Brady has 111, NMD has 66). Each blower logs three values:

- The temperature set by the user [°C] (only for Brady oven).
- The measured temperature [°C].
- The output power at the solid-state relay of the reflow.

Random inspection of this huge dataset showed that usually only a subset of blowers logs significant data, while the others report zero, negative or out-of-range values.

Event files are also provided. They are referenced one per day, matching one to one the data files. Each event file contains a list of logs related to the oven. Each event has a timestamp and a textual description. Key to the predictive maintenance scenario are the failures of each blower. Unluckily, the event logs do not specify which of the blowers failed. Furthermore, at a purely ballpark analysis, the number of faults seems to be unbalanced compared to the number of samples. In details:

- Brady → 15 failures.
- Tachy → 0 failures.
- Rhythmia → 1 failure.
- NMD → 7 failures.

The data files globally contain 2725344 samples distributed as follow:

- Brady → 649152 samples.
- Tachy → 652032 samples.
- Rhythmia → 366912 samples.
- NMD → 942048 samples.

Additionally, BSL provided an excel table of blower failure records. Each failure has:

- A numeric ID, assumed to be the blower's ID.
- The description of the intervention (which is always the substitution of the blower).
- The intervention timestamp.
- The name of the oven the blower belongs to.

The Figure 8 shows some plots of the Brady oven blowers corresponding to faults. The colour meaning is the following:

- Blue plot is the temperature set by the user.
- Orange plot is the temperature measured.
- Green plot is the output power.
- Red plot marks the points where faults occurred.

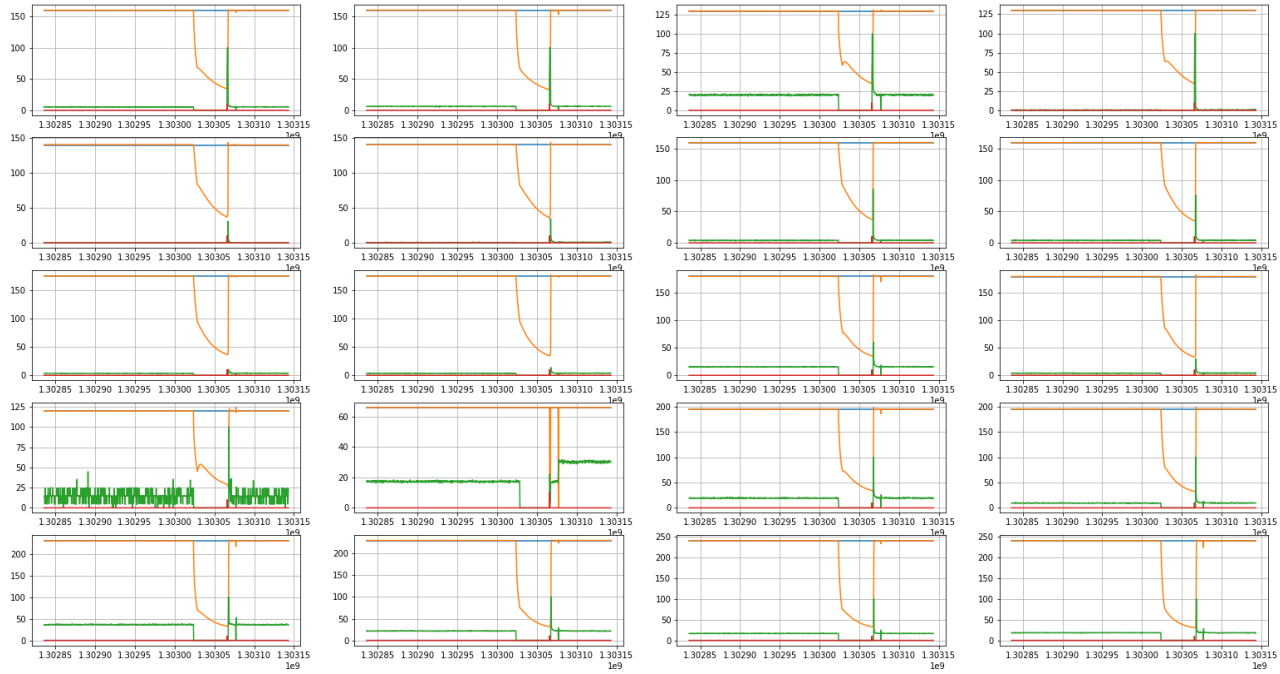

Figure 8: Brady oven dataset from 04/15/2011 to 04/18/2011

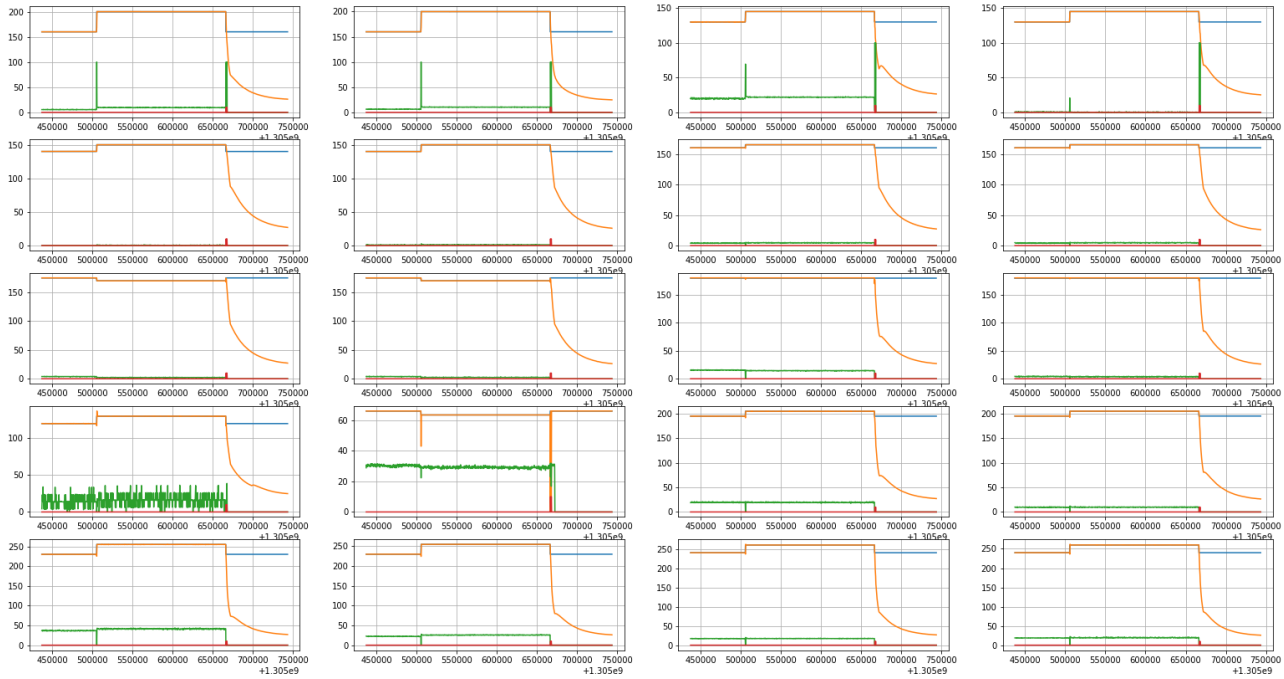

Figure 9: Brady oven dataset from 05/15/2011 to 05/18/2011

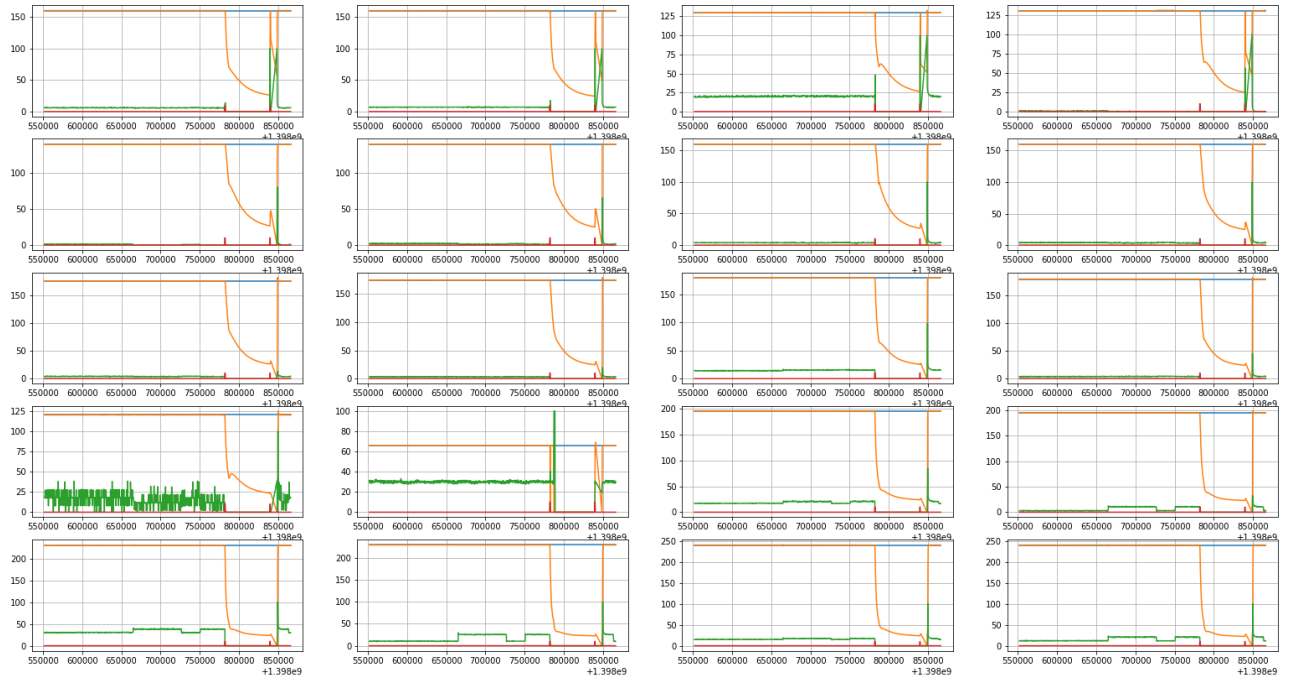

Figure 10: Brady oven dataset from 04/27/2014 to 04/30/2014

It is worth noticing that, for both Figure 9 and Figure 10, it doesn't seem to exist or cannot be seen with the naked eye a characterization pattern that makes an oven's fault prediction possible. In fact, unlike what happens in Figure 8, where the measured temperature (orange line in the plot) steeply decreases, dropping to zero before the recorded fault event, in the other plots the temperature starts decreasing only after the fault event. This is an asymptomatic issue of the input data, which steepen the problem curve tackled by the DLT, and highlights a trend of non-correlation to the problem class that the component is aiming to solve.

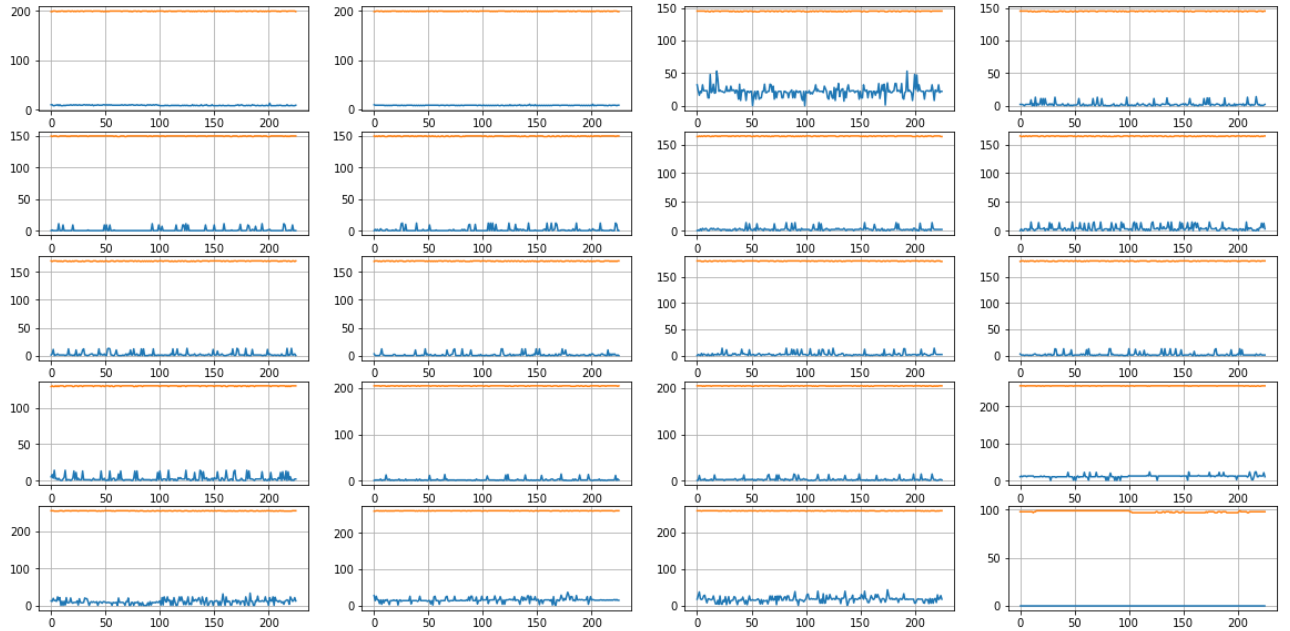

Figure 11: Rhytmia dataset on 07/15/2014

The above figure, instead, shows the value of Output power (blue line) and Temperature measured in C° (orange line) of the Rhythmia oven during one day where the BSL operators reported a blower failure.

It is necessary to point out since Rhythmia does not log the fault automatically an operator has to report on a separate file the occurred faults. Since the reported failure does not record the hours, the failure occurred at, it is impossible for us to show on the plot the exact moment when the failure takes place.

6.1.2.2 Acoustic data

The term Acoustic data refers to a new type data obtained after the work of Tyndall of studying, developing, deploying and processing the data from the acoustic sensors placed inside the oven.

The data refers to five acoustic sensors, which measure the noise level of the oven every five minutes. The use of acoustic sensors comes from the experience of the operators of the BSL shop floor who use the noise produced by the blowers to decide if a fault is happening.

Each line of the processed file has the following structure:

- Date in the format: year, month, day, hour, minute, second
- Sensor's reading

TYNDALL provided the data in five different files one for each sensors.

A complication with the acoustic data is that, at the time of writing this deliverable, the amount of data provided by the partners is neither enough to train a network nor to be used as a feature for an already created model.

Details of data at the time of writing:

- 66407 incomplete records (at least one sensor has data)
- 529 complete records (exactly one read for each sensor)
- Complete data from: 2018-05-23 14:33:29 to 2018-05-25 09:44:00 (2 days)

Below is a plot of the previously described (complete) data set:

Figure 12: Acoustic sensors 05/23/18 - 05/25/18

there are five different sensors deployed in the shop floor. One problem that easily arises from the previous graph is the lack of one sensor. At the time of writing this deliverable there is an anomaly with the data: the read of one sensor was overlapping with the other.

6.1.3 Fitness for usage

6.1.3.1 Legacy data

The dataset type and size is suitable for the predictive maintenance task. Still some major issues remain. The number of failure events is way too low: the best-case oven has 15 failures over more than 500K samples. As a consequence, the sampling frequency results are too high and the single samples become irrelevant which results in an unbalanced situation where sample aggregation and under sampling are inevitable. The dataset cardinality will be therefore strongly reduced, which will affect its usefulness, based on the number of discarded samples. Moreover, while data samples report measurements on a number of blowers (typically 6), the failure

events do not provide a lot of information for identifying the damaged blower. This may affect the capability to correctly predict future failures.

Other problems arise from the different nature of the datasets (Brady and Rhythmia), the entire preliminary study has been done on the Brady and therefore only some results are partially true. The biggest differences lie on the nature of the two datasets and their dimensions. The amount of data (~650000 samples for Brady ~370000 for Rhythmia) makes Rhythmia's dataset half of Brady's one; Moreover, the number of faults (15 for Brady – 1 for Rhythmia) resulting in 1 fault every ~43000 samples for Brady and 1 fault every ~370000 samples for Rhythmia marks an even more unbalance of the dataset. The nature of those faults also changes, in Brady the faults are automatically logged by the oven and coincide with an actual break on the oven's blower, while in Rhythmia are written (not always) in a file by an operator. This means that there is not an automatic alignment of the fault and the data, resulting in the obligation of switching to a new approach.

To cope with this short amount of faults, after talking with the operators from BSL, a pattern of faults has been identified. An operator usually checks the logs and changes the blower (he decides it is a faulty one) in two cases:

1. When **three** events of type "**Hi warning**" are presents in the previous thirty minutes **outside the oven stabilization** (it has been defined an oven stabilization period the thirty minutes following the loading of so called "wakeup recipe").
2. When **one** event of type "**Hi warning**" plus **one** event of type "**Hi Deviation**" are present in the previous thirty minutes (at any stage)

The dimension and the quality of the Rhythmia' data make it indeed challenging to build a Deep solution model with them.

6.1.3.2 Acoustic data

The preliminary analysis has brought to light some interesting patterns in the acoustic data provided. This analysis was conducted exclusively on the data themselves, because due to their quantity it was not possible to carry out a full analysis correlating the acoustic data with the legacy data. The lack of correlation does not prevent the use of the data, but it prevents it from giving a definitive answer on the suitability of the information in the use case. Nevertheless, the data fit all the empiric criterions for a deep learning model.

6.1.4 Data assessment

6.1.4.1 Legacy data

Given the provided information, the best possible approach to predictive maintenance use case is to train a LSTM Neural Network (the state of the art of Recurrent Neural Networks) on the data related to Brady oven, which provides the highest number of failures. Subsampling/aggregation of data is indispensable. The other criticalities to deal with are the large quota of missing/invalid data and the absence of correlation of failures and blowers. Because of this, it is not straightforward to tell beforehand whether the trained model will achieve an acceptable prediction accuracy.

Continuous learning on live data is expected to relieve the low accuracy problems over time, as long as the data provided by the oven blowers will be valid in live situations.

6.1.4.2 Acoustic data

As described during the present section is not possible to draw any conclusion regarding the use case. The volume of data, at the moment of writing the deliverable, 2 days of complete data or 529 records (for detail see 6.1.2.2) is insufficient for modelling any solution apart from trivial ones. A complete analysis and modelling of will be done upon receiving the full data from the year 2018.

6.2 Prices and logistics (UC-ELDIA-1)

6.2.1 Background

This use case is the only one listed in this chapter that is not related to the intra-factory scenarios, but instead is more related to the inter-factory environment. The Deep Learning Toolkit component is expected to distribute

the latest prediction on the price per ton at which specific commercial partners are likely to accept to buy/sell scrap metal within fixed timeframe in the future. This information in the form of predictions are intended to support the agent intelligence in order to improve the decision system that is in charge of accept/emit commercial offers about scrap metal.

6.2.2 Data Overview

Both the end users involved in this use case (Kleemann and Eldia) have contributed providing data about waste management.

Kleemann provided a minimal scrap metal dataset containing only 12 samples. They are equally distributed: one for each month of 2016. Each sample has 11 features, including the quantity of 8 different metal scrap types and of 3 other materials (plastic, wood and paper) produced along one month (measure unit is not clear). A consistent part of the data is missing.

Eldia provided its historical data in the form of four excel tables from which two datasets can be extracted. The first is related to transactions on scrap metal whereas the second is related to transactions of other materials.

The scrap metal dataset accounts for sales and purchases: each record summarizes the transactions between Eldia and one commercial partner over a single month. In particular, the dataset contains 144 samples sales samples (3 clients x 12 months x 4 years) and 192 purchases samples (4 suppliers x 12 month x 4 years) accounting for the period [2013-2016]. Each sample has six features:

- Type of transaction.
- Timestamp.
- Client id.
- Scrap metal quantity (ton).
- Number of trips for collection/delivery.
- Price per ton.

The dataset relative to other kind of waste contains records of purchases aggregated by month for the period 2015-2016: 192 samples (4 suppliers x 12 months x 2 years). Each sample has 10 features:

- timestamp
- suppliers' id
- wood: waste quantity (ton)
- wood: number of trips for collection/delivery
- plastic: waste quantity (ton)
- plastic: number of trips for collection/delivery
- paper: waste quantity (ton)
- paper: number of trips for collection/delivery
- general waste: waste quantity (ton)
- general waste: number of trips for collection/delivery

Considering that only one of the suppliers provides all waste categories, the dataset has a considerable amount of missing data.

Together with the above-described data, ELDIA provided another dataset for a different materials. The dataset contains information about recycled paper prices together with eight prices each one for each type of paper. One record of this dataset contains the price per ton for the following type of paper:

- Mixed paper and board
- Supermarket corrugated paper and board
- Old corrugated containers

- Sorted graphic paper for deinking
- Unsold newspaper
- Coloured letters
- Multi printing
- White wood free uncoated shavings

The dataset contains one price (for each category) per month this in a time span that goes from 01/01/2017 to 01/12/2018.

6.2.3 Fitness for usage

None of the provided dataset is suitable for training a price-based prediction model. Appropriate data would be a data set that includes, for each type of waste and each commercial partner, a time series of waste offer/transaction prices covering the widest possible period. The scrap metal dataset by Eldia is close to fit the aforementioned format, but the number of samples is excessively small and the price trends is negligible. Moreover, the results of transactions under long-term partnership prices are almost constant: adjustment only happens on a yearly basis. Despite this being very understandable from the commercial point of view, it precludes the fitness for usage in a dynamic marketplace scenario that encompasses frequent price fluctuations due to constant negotiations.

6.2.4 Data assessment

Eldia is updating the prices over time and recording every fluctuation. The data are aggregated sales statistics spanning the first three quarter of 2017. All records relate to the same customer, but differentiate in terms of material type (paper, PET, HDPE, scrap metal) and timespan. The samples have the following features: material type, date start, date end, tons, price per ton. Indeed, the data contains price fluctuations, which make more sensible to train predictive models on them. Still, the number of samples is very limited (ranging from two samples for PET to 16 records for paper), at least three orders of magnitude too low to perform a significant training.

Detailed action plan on how this problem was addressed is presented in chapter 7.1.

7 Deep Learning Toolkit design

The goal of this chapter is to streamline the different results that have been accomplished in the second iteration of COMPOSITION (M30).

All the improvement reported in the following sections had taken advantage of the insights gained during the first iteration of the project (M16). To increase the readability of the document, all results collected previously have been moved in Annex I section 10.2. Nevertheless, cross-references have been created directly inside each chapter to provide an easy recovery to the most relevant information.

This indeed is the central chapter of the deliverable where the design of the DLT will be explained in detail and in each of its part. The two different instances of the DLT (one for each use case) will be treated as separated from the other to simplify the reading of the document.

First a comprehensive description of the DLT for the use case of the price prediction followed by in depth characterization of the DLT for the predictive maintenance, respectively: chapter 7.1 and chapter 7.2.2.

For the price prediction DLT together with the proposed and deployed solution, it will be described an advanced experiment which uses more data to improve the performance of the deployed model.

For the use case of predictive maintenance together with definition of the model using the legacy data, new sensors have been deployed recently on the Rhythmia oven providing new acoustic data stream. Extensive research on frameworks and technologies have been conducted to integrate this information in the Deep Learning Toolkit models. The chapter 7.2.3 describe the analysis carried out on the first batch of acoustic data.

7.1 Supervised learning results on price prediction from UC-ELDIA-1

As described in section 6.2, Eldia is providing the prices over time and recording every fluctuation. The data are aggregated sales statistics spanning from 2016 to 2018. All records relate to the same customer, but differentiate in terms of material type (paper, PET, HDPE, scrap metal) and timespan. The samples have the following features: material type, date start, date end, tons, price per ton.

Predicting prices of raw material is a very tough task for ML algorithms. Indeed, usually in these cases, there is a lack of pattern and there are many parameters affecting it, which cannot easily be listed. It is not easy even for a subject expert to determine which variables are affecting it. There may be factors related to geopolitics, economics, other good prices, etc. Some of them, moreover, are nearly impossible to monitor, e.g. laws, rumours caused by someone's declaration are very hard to track and detect. Without knowing all these factors, it is impossible to estimate correctly the goods prices. Some preliminary results with data from a real marketplace have been obtained in the first iteration of this document and are available in Annex I section 10.2.4.1.3.1.

Better results can be achieved if it is possible to identify any recurrent prices trend tendencies as demonstrated by results obtained with univariate synthetic time series in Annex I section 10.2.4.1.3.2. At the moment, the number of samples for real use-cases is still very limited (ranging from two samples for scrap-metal to 16 records for paper) because prices adjustment only happens on a month basis and price trends are negligible. However, two different multivariate approaches have been evaluated leveraging on continuous learning capabilities (section 7.1) or involving additional features obtained from French market (section 7.1.2).

7.1.1 Eldia goods prices estimation with continuous learning

In order to produce a reasonable amount of data for training the network, ELDIA's datasets has been resampled on daily basis. This was needed for improving the quality of the dataset, since the provided data had one entry per month. The resulting datasets for paper end scrap metal goods are showed in Figure 13: the blue plot represents the price trend of paper and scrap metal goods in euros while the orange plot is the daily amount of good exchanged in tons, the green plot is an artificial feature that will be explained afterwards.

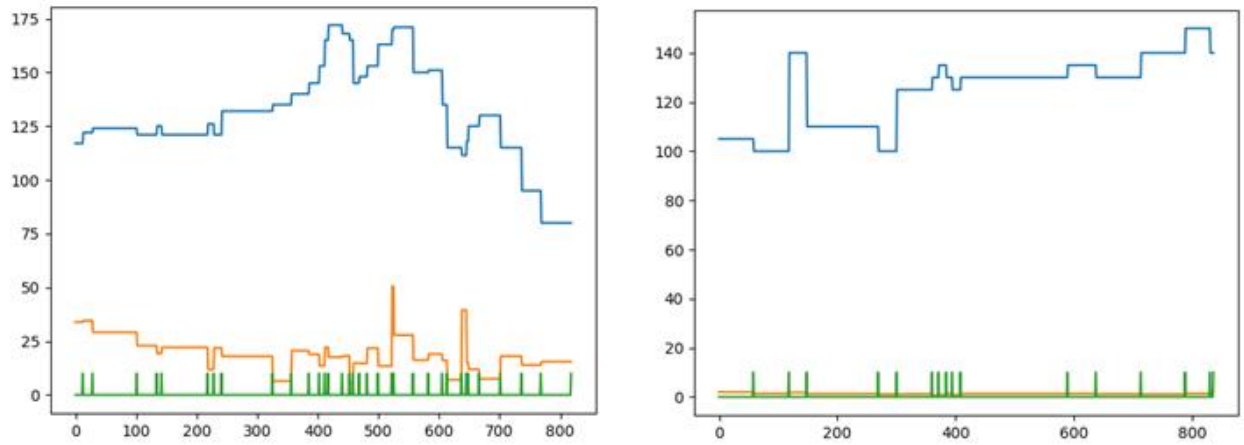

Figure 13: paper (left) and scrap metal (right) input data set

The results of single-point prediction models based on historic price data alone are hard to accomplish. Normally, the network seems effectively able to learn the price trend, but it often predicts a value equals to the value at the previous time point, though it still minimizes the loss function. This behaviour, named “mimicking”, is very harmful especially with dataset composed by sequential constant size steps like the ones described in this section.

The third feature (green plot in figure) is a synthetic one to identify when a price variation occurs. It is a higher-level representation of the raw dataset that helps to reduce the mimicking effect teaching the network when a price variation happens. Furthermore, the network architecture and hyper-parameters has been tuned to cope with this particular issue. However, the best way to improve the results would be to add features that go beyond historic prices alone.

The raw values are converted into samples with the procedure described above:

- n_ts_in = 64 timesteps
- n_ts_out = 1 timestep

The initial network for regression is composed of four hidden LSTM layers with 64, 32, 24 and 8 neurons respectively and hyperbolic tangent as activation function.

Figure 14 shows the training result for scrap metals after 35 epochs:

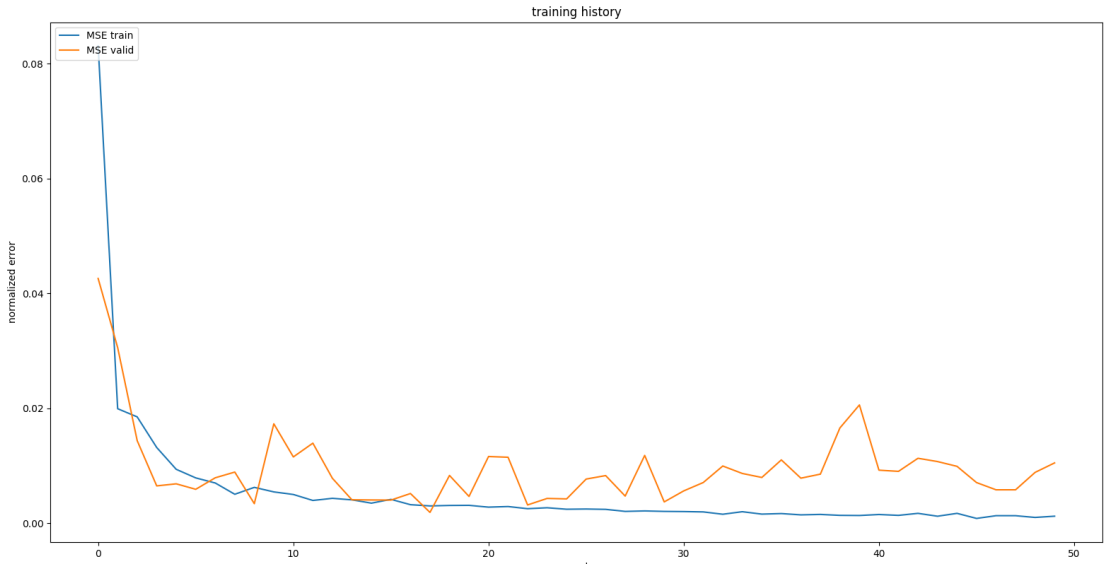

Figure 14: scrap metals training results

Both the blue line, corresponding to the normalized error on the training data, and orange line, corresponding to the normalized error of the validation data, show a clear convergence is rapidly achieved after very few

epochs. Its fast convergence often leads to poor generalisation performance because the ANN has been trained using too few constant step samples and the network overfits, learning all available features by memorizing the training dataset in its entirety.

This is evident in figure below: it shows the overfit increasing, regarding the accuracy, in relation to the epoch number.

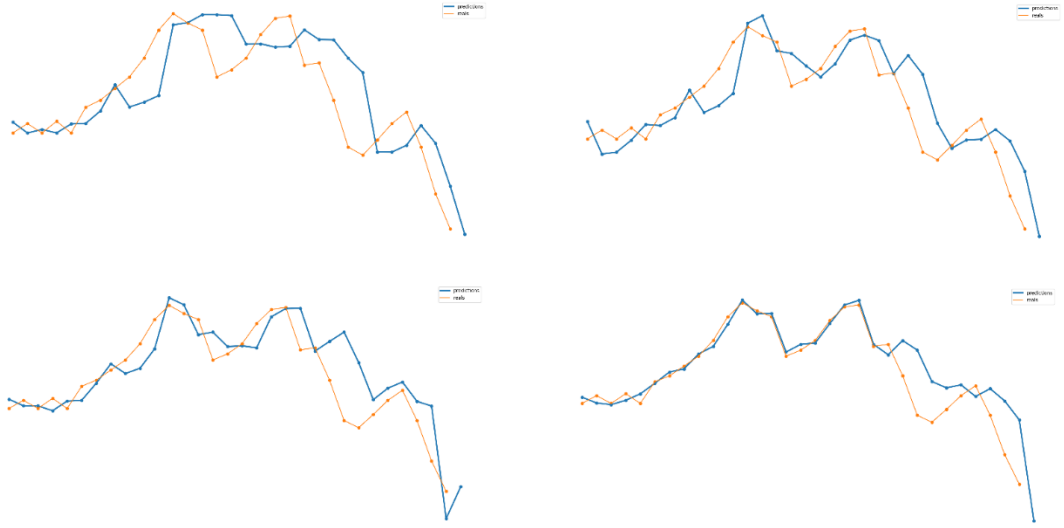

Figure 15: overfitting phenomenon with epoch increasing on paper good. Top left 10 epochs, top right 50 epochs, bottom left 100 epoch, bottom right 300 epochs

It is clear that the network overfits because the two plots, orange and blue, overlaps always better on the training data (left part of each graph) with the increasing of the epochs. Results on the validation datasets (right part of each graph) are predicted with less accuracy. To mitigate the phenomenon the number of epochs was kept low intentionally.

The network actually deployed has been trained on 50 epochs. Its prediction results are available in the figure below:

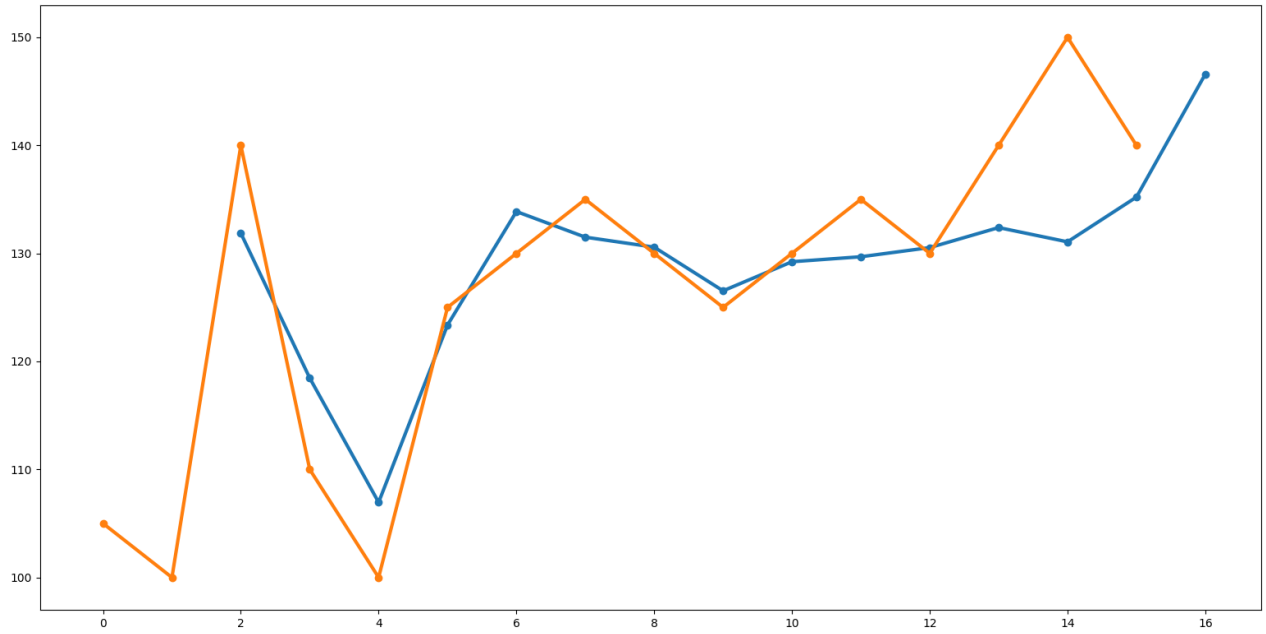

Figure 16: trained network (50 epochs) results for scrap metal

Figure 17 shows the results obtained when the network starts completely untrained leveraging only on continuous learning capabilities for training. Initial results are extremely inaccurate because the network is predicting the results using random weights. Its accuracy improve as additional data become available.

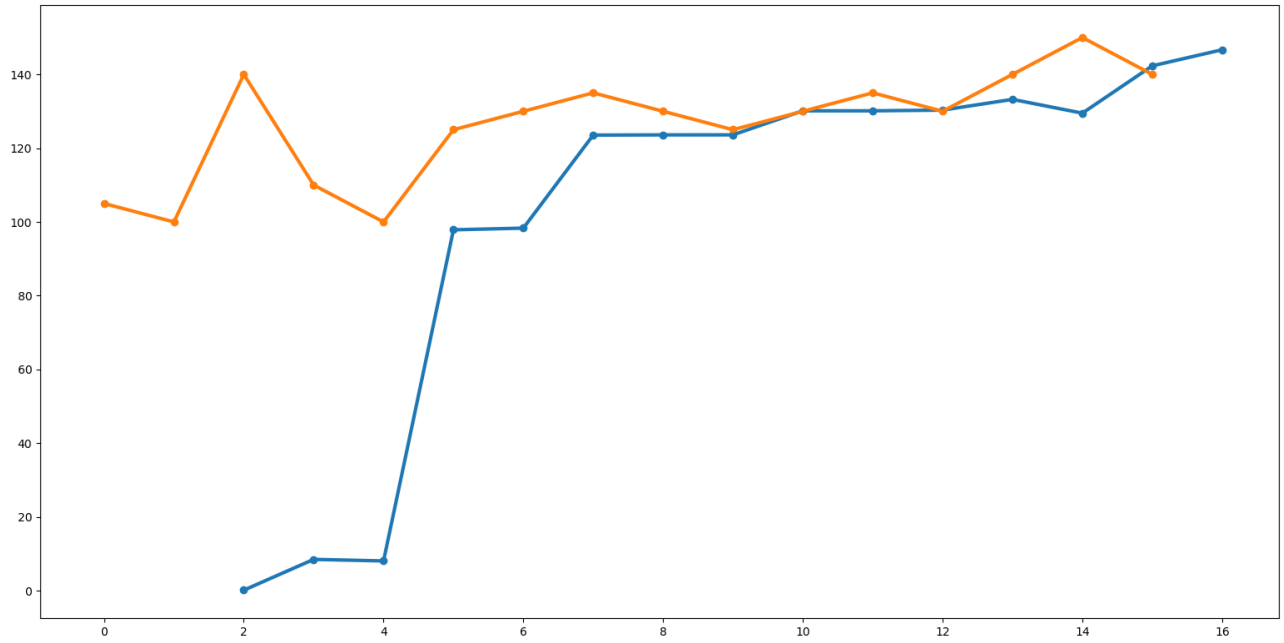

Figure 17: untrained network results for scrap metals

Unfortunately, it is almost impossible to understand the long-term implications on the network behaviour because they are strictly dependant on the input data that will be provided to the network. But it is safe to assume that, if the boundary conditions do not change from the time of obtaining the data to the time of using new data, the network will learn the price's pattern with the help of continuous learning. That means that if the boundary conditions when the network have been trained are the same of when they will be used to predict or to apply continuous learning the network will converge.

In fact, it has been demonstrated with synthetic data in section 10.2.4.3 that a convergence is possible in some specific scenarios and could happen in a finite time span.

7.1.2 Eldia goods prices estimation with additional features

In order to improve the poor accuracy of the previous results, one solution that has been tested was enriching the incoming data with the information provided for similar items by other markets. Despite these concerns are not completely addressed by this approach, the behaviour of different kind of paper price is likely to be correlated, so the information may help the model to understand general trends.

The test had been performed on the prices of paper together with the ones of different type of recovered paper in France by EUWID, in order to predict the future prices in the platform. The paper has been chosen, despite it is out of scope of the use case, because the partner had the higher amount of data on it.

There were 8 different types of recovered paper and for each of them there were the minimum and the maximum price in € per tonne for each month, for the range between January 2017 and December 2018. Please, notice that despite the time range is not very small, the amount of data is indeed very little: each dimension has only 24 samples. In the same timeframe, the number of different prices in the platform is 28. Since one of the key reasons of AI success nowadays is the availability of large datasets, it is evident that our work has been highly affected by the lack of a significant amount of data. The figure shows the input features: values do not vary too much and their trend is not homogeneous.

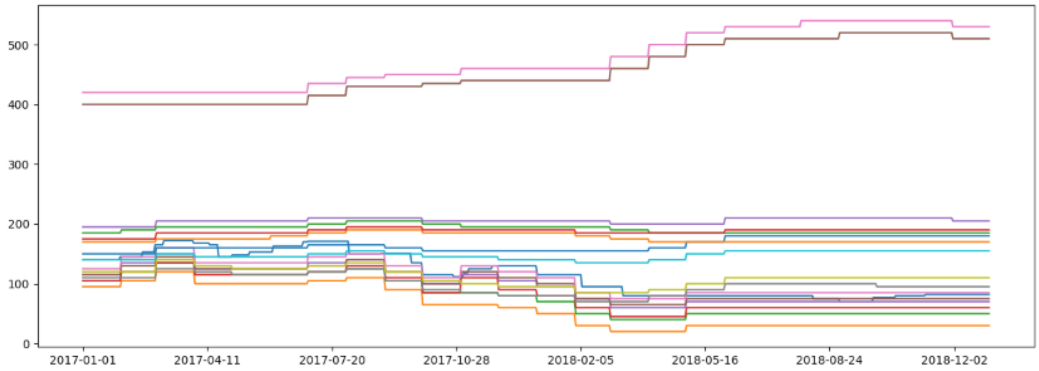

Figure 18: input features

The first operation performed was resampling the data on a daily basis, in order to produce a reasonable amount of data for training the network. This does not address, though, the problem exposed in the previous paragraph. The model was trained using a window size of 15 for the input and 30 for the output: it means that the model uses 15 days of data in order to predict the behaviour of the prices in the next month (30 days). All the values have been scaled using standardization.

The RNN built for this experiment was made of two main building blocks:

- Encoder: the encoder layer is mainly composed of a LSTM node.
- Decoder: the decoder consists of a LSTM layer, followed by a fully connected one, which produces the output (through a linear activation).

Since the goal was to predict the future prices, the metrics used in order to evaluate the results have been the MSE and the MAPE.

In order to prevent the RNN to over-fit too much (despite collecting more data is the only way to make the model able to generalize properly), the following techniques have been applied:

- Adding dropout;
- Adding L2 regularization on the weights and activation parameters, L1 demonstrated to work poorly in this use case;
- Early stopping, in order to prevent over-fitting on the training dataset when this brings no value on the validation set;
- Adding more layers and making the network deeper had no effects on improving the results. It means that the capacity of the abovementioned model is enough to describe the problem.

The hyper-parameters of the model have been chosen with a combination of manual tests with some provided values and runs leveraging the hyperopt library (Hyperopt, n.d.) in order to determine the best ones for the specific use case. The considered hyper-parameters are the learning rate, the dropout percentages, the batch size, the kernel regularization factors, the activation regularization factors. Tuning them all is critical to get reasonable results.

7.2 Supervised learning results on predictive maintenance from UC-BSL-2

This section describes in detail the results of the effort on the use case of the predictive maintenance using the data coming from the ovens in the shop floor of Boston Scientific. The section is divided in two parts - one for Brady oven and one for Rhythmia - because the data of the two ovens and the approaches are completely different. Both the subsections follow the same structure: first, a brief description of the dataset creation; then, a wide description of the technique used and finally the result of the proposed solution.

7.2.1 Brady oven

The section explains in detail the approach that have been taken to tackle UC-BSL-2 on the data of Brady oven.

The dataset has been created merging the data files and the event files using the timestamp as key. In detail, iterating all the files in pairs (data, event) for all the days for which there are measures. Since the quantity of events and data entries are not the same, this means there is not an event for each data entry or vice versa. It has been decided to “merge” the two data (logs + measures).

It has been chosen to assign for each data entry, in the data file, the event with the closest timestamp, duplicating the events where needed. The process is described in the following figure.

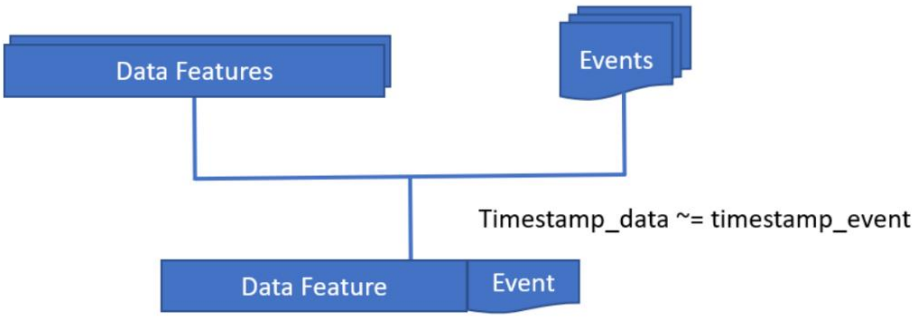

Figure 19: Brady dataset creation

As show in Figure 19 the process consists in creating one single entry, associating for each data entry its event. The reasons why this approach was chosen are: there are more data entries than event entries for each file; the assumption that if an event occurs at the time (t), there is a high probability that the same event “was occurring” or about to occur at the time (t-1) or (t+1) since the time between two data entries is close enough (5 minutes).

The second step was substituting the feature “Set Value” (SV), for each zone of the oven, with the difference between “Set Value” and “Present value” (PV). This was done because “Set Value” defines the value of temperature set by the operator of the oven and it turns out to be constant within a time interval. We hence decided to substitute it with the difference between the SV and the PV in order not to lose the information about the “Set Value” but without inserting an, almost always, constant feature in the model.

The last step in the creation of the dataset was translating the event text in a categorical value (0, 1, …, N). We decided that the following class of event are the most interesting along all the possible ones:

- Flux Heater High Warning
- Hi Warning
- Hi Process
- Lo Warning
- Hi Deviation
- PPM Level within
- PPM Level has exceeded the amount set
- High Water Temp Alarm Cool Down Loaded
- Low Exhaust Alarm
- Exhaust is insufficient
- Heat Fan Fault
- Blower Failure (Fan Fault).

NB: the last two event types are actual fault of the blower of the oven.

Assigning a value from 1 to 12 to the events in the following list, 0 otherwise. Summarizing: when the new feature called, “Log_num” has value of 0 means no Error (or not interesting event). Instead, when “Log_num” has a value between 1-12 means the correspondent event in the previous list occurred.

The ideal Deep Learning Toolkit would predict, based on the previous data, when a fail will occur. In order to achieve this result after studying the state-of-the-art and most innovative models of time series classification/forecasting and comparing with the well-known ones, we decided to choose the Recurrent Neural Network (RNN). Because each read of the sensors and each event can be seen as discrete variables that changes through time.

The data is fed to the network in **consecutive time series of length 32** and the network will predict the following (starting from the last point of the time series) **5 future events**.

One sample of the training dataset is defined as follow:

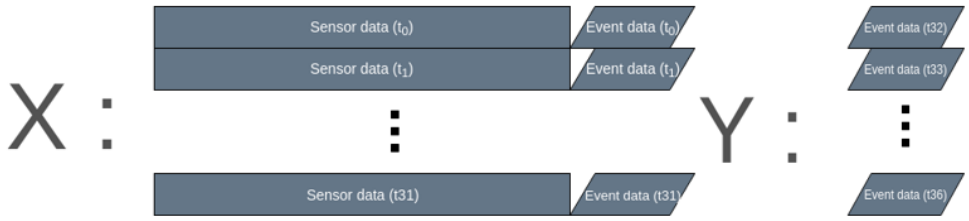

Figure 20: Brady training data

Regarding the data format in the Figure 20, it is necessary to specify that the events (in both input and output) will be passed as One Hot Encoding (OhE).

Summarizing the whole procedure: the network will take a time series of length 32 for each one of the 73 features (sensors + events) and will try to predict the future 5 Ohe serialization of the events, hence predicting a [5x13] matrix.

Simplifying the network as a function f(x):

$$dim(X) = [32,73]$$

$$f(X) = Y$$

$$dim(Y) = [5,13]$$

Before detailing the architecture, it is necessary to explain which kind of RNN we decided to use. The inputs are **73 (feature dimension)** time series of length 32, and we want to output **13 (output dimension)** time series of length 5. The correct type of RNN to apply is many-to-many with of T_x (input) and T_y (output) with different lengths, this approach is shown in the network number 4 in the following picture:

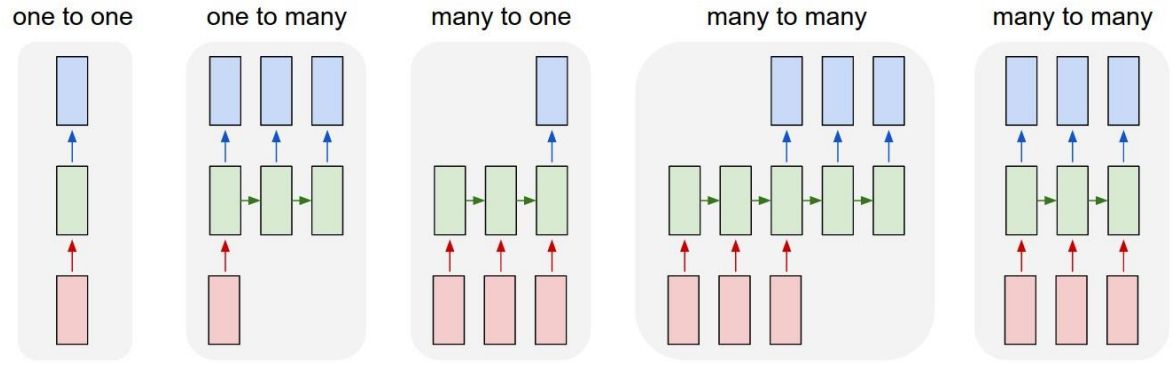

Figure 21: Different Type of RNN (Karpathyc, 2015)

The detailed architecture of the network is the following:

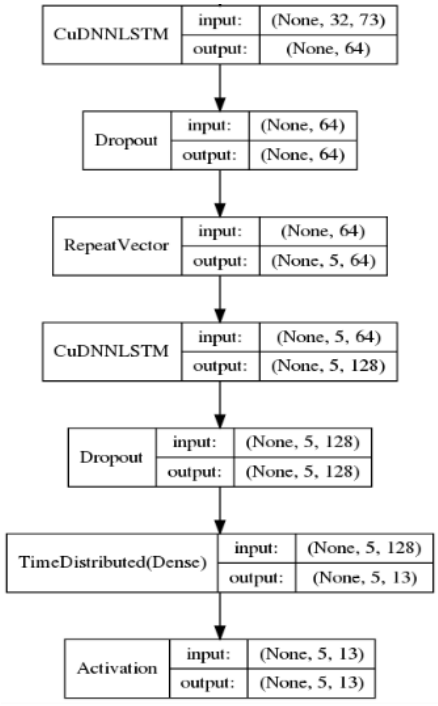

Figure 22: Network architecture

The network consists in an encoder and a decoder stacked on each other. The encoder consists in the first LSTM with 256 units while the decoder is the last LSTM with 64 units as well. The first layer takes in input 32 time steps for the 73 features and returns only 64 vectors representing the "encoding" version of the input data. These encoding data is repeated 5 times (the number of time step in the future we want to predict) by the Repeat Vector layer creating a new time series of [5x256].

This newly created times series is given in input to the decoder, the second LSTM. The decoder returns the new 5 values plus their "sequence": this can be seen as a time series (of [5x64]) too that passes thought a Dense Layer in a "time distributed fashion". It means, citing Keras documentation, that "This wrapper applies a layer to every temporal slice of an input".

The last layer uses the softmax activation: this is obvious because we have an OHe as target variable (Y).

The architecture in the Figure 22 is the one that achieved the best performance among all the other possibilities.

The final architecture is the output of several tests on different ones. In the following figure, it is possible to see all the possible hyper-parameters we tested the network on.

```
{
        'units1': hp.choice('units1', [32, 64, 128, 256, 512]),
        'units2': hp.choice('units2', [32, 64, 128, 256, 512]),

        'dropout': hp.choice('dropout', [0.25, 0.30, 0.50]),
        'shuffle': hp.choice('shuffle', [True, False]),
        'batch_size': hp.choice('batch_size', [16, 32, 64, 128]),
        'lr': hp.choice('lr', [0.01, 0.001, 0.0001, 0.005, 0.002]),
        'reg': hp.choice('reg', [True, False]),
}
```

Figure 23: Hyper-parameters space

The only options that need to be explained are: 'lr' stands for learning rate; 'reg' defines whether to use or not the 'amsgrad' (S. J. Reddi, 2018) optimization for the Adam optimizer (D. P. Kingma, 2015).

NB: the metric used as loss function is the categorical cross entropy (Entropy, n.d.)

Following will be detailed the two techniques used for preventing the overfitting: First, the dropout, which is a special layer in a NN that has the function of randomly "turn off" one random neuron from the previous layers and readjusting all the weights on each training step. The percentage of the "random" can be set and usually has value >= 0.5.

The last technique used for preventing the over-fitting is the so-called “Early Stopping”. This technique allows training of the network until the validation loss stops improving. Of course, there are some epochs of “waiting” before stopping the training in order to allow the network to keep learning even after some not-learning epochs.

The time needed for creating the dataset starting from the raw data files is about 16 min, while the time for loading the already processed dataset before training is 38 sec.

The time for training the network on 1000 epochs is 278 minutes, ie. 4 hours and 38 minutes, hence more than 16 seconds per epoch.

The data have been split in 80% training 10% validate 10% test, due to the high unbalance of the classes. During the training, we have used the validation dataset to evaluate the accuracy of the model and to choose the epoch with the best validation loss.

The weights of the model at the “best” (according to the above definition) epoch will be the final weights of the model which will be used for predictions.

The training of the network shows an accuracy of **0.87115** and a precision of **0.86620** on the test.

The **precision** or **Positive Prediction Value** (Wikipedia, n.d.) is the fraction of relevant instances among the retrieved instance:

$$\mathrm{PPV} = \frac{\text{number of true positives}}{\text{number of true positives} + \text{number of false positives}} = \frac{\text{number of true positives}}{\text{number of positive calls}}$$

Equation 1: Positive Predictive Value (PPV)

Following will be shown the plots of the metrics, obtained during the training of the network.

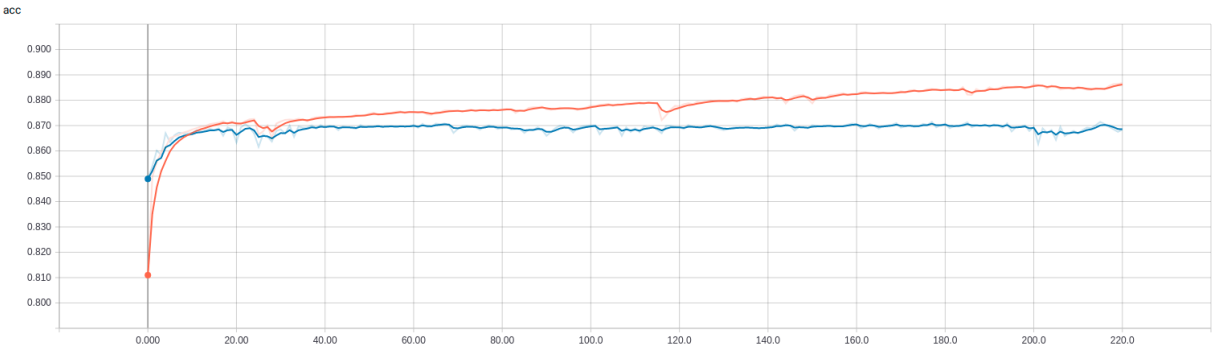

Figure 24: Accuracy plot

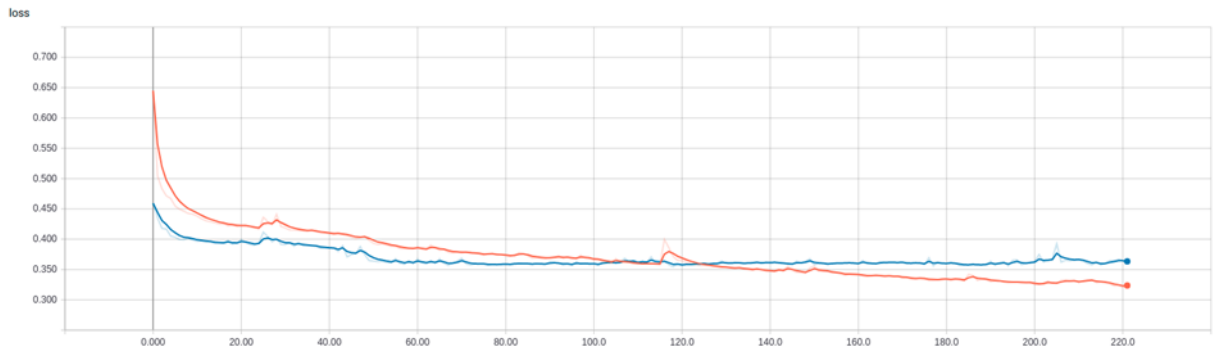

Figure 25: Loss plot

In the Figure 24 and Figure 25 are shown the accuracy and the loss during the training phase (orange line training dataset, blu line test dataset), it is possible to see that the slope of the loss in gentle meaning the learning rate and the number of epochs are likely the correct ones.

It is also interesting to focus on the “small step” in the loss: they depend on the “amsgrad” parameter of the optimizer.

In detail the results of the network on the test-set.

Figure 26: Confusion Matrix

In the Figure 26 is shown the Confusion matrix of the “first event in the future” that means the first event that will happen in the next 5 minutes (starting from the input time series). Of course, this prediction is the less useful but the most accurate since the network does not have to “see” too far in the future.

The diagonal of the matrix indicates the correct predictions for each class. The only class whom does not have a good overall performance is the High Deviation: with 83% of times predicted as no event and 17% as a High Warning.

These poor results can be partially explained by the low occurrence of this event in the dataset, where the High Deviation has only 1621 occurrence in the whole dataset. While 6 in the test dataset; Zooming-in the confusion matrix not normalized:

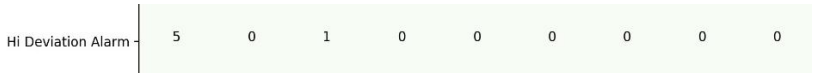

Figure 27: Zoom of the confusion matrix

Another useful graph that can be shown are the Roc curves.

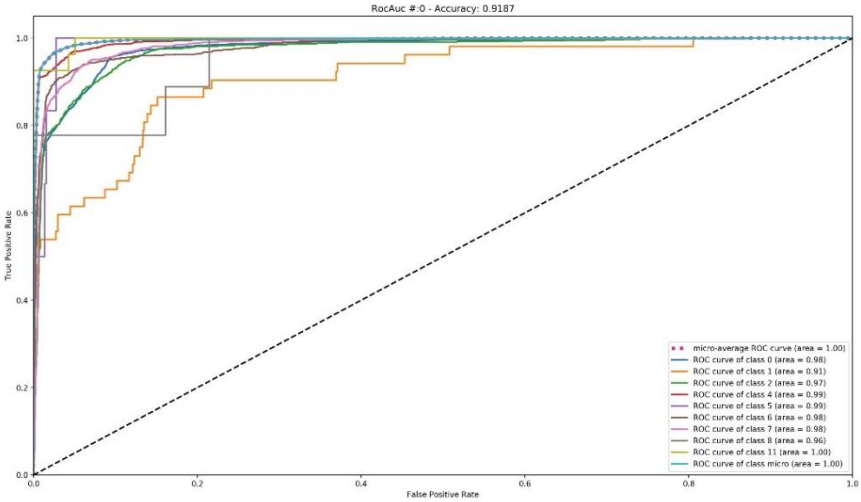

Figure 28: Roc and Auc curve

The Figure 28 shows the ROC curve of the UC-BSL-2 classifier. A ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters: True Positive rate and False Positive rate for each class of the dataset. The purple dotted line shows the curve using the micro-averaging. With this plot we can choose the threshold to which decide the class of the event.

Before the conclusion, will be shown a test trying to predict more than 5 elements in the future. For accomplish this tests it has been necessary to slightly change the procedure of creating the dataset described above, indeed the matrices change regarding the amount of time steps in the future the network is trying to predict.

The test settings are the following: the network predicting the future 10, 15, 32 events in the future. Remember predicting:

- One event in the future -> next five minutes.
- Fifth event in the future -> next twenty-five minutes.
- Thirty-two event in the future -> next 2h and half.

Table 8: Table test more events

Future Prediction (min)	Accuracy	Precision
30	0,77810	0,77830
15	0,80646	0,80556
10	0,83281	0,82725
5	0,87115	0,86620

The trivial supposition that increasing the “time in the future” would decrease the performance turns out to be valid. The model shows a good overall performance to predict the next 5 events in the future or with less accuracy the next 10.

7.2.2 Rhythmia oven

During the second iteration of the project (M22), the input dataset has been updated by integrating new features that further developments made available. These changes have been to a large extent largely originated in the switch from Brady to Rhythmia oven for BSL-2 predictive maintenance scenario.

The base data described above have been adapted by filtering logs events provided by the oven (e.g. temperature warning, deviation alarm, etc.).

As described in section 6.1, there are some differences in the sensors and logs produced by the two ovens. These differences (see 6.1.2) make hard or impossible to adopt the previously trained model. The main reasons affecting the efficiency and usability of the Brady ANN model (7.2.1) are:

- The lack of the “temperature set by the user” feature for each oven zone (in total twenty features are missing);
- The lack of automatically logged faults, replaced with a pattern of events that the operators of BSL refer as a fault.

On the other hand, the main challenge in developing a completely new model tailored for the Rhythmia oven is the detection of oven failures and faults. Unfortunately, Rhythmia logs do not provide enough information to compute efficiently these features. Moreover, the excel table provided by BSL reports only two blower failure records in the period from 2012 to 2018. These records are not supplied with a time (only date) making impossible to accurately position of the event in the day.

Consistently with these dataset limitations, the prediction model evolved taking in account suggestions from the BSL oven operators to better identify failures causes and to improve the quality of the predictions.

In this specific application context, the input dataset is represented by a collection of oven logs that need to be appropriately characterized in order to understand which are the features that allow to discriminate a correct oven behaviour from a faulty one.

The resulting comprehensive dataset, has been identified in order to create a suitable mapping of all the available information to be provided to the DLT. In specific, a table of 242 columns was aggregated and qualitatively and quantitatively characterized. Each table row can be split in four parts as shown below:

0	1	2	3	…	39	40	41	41	43	44	45	…	234	235	236	237	238	239	240

Table 9: DLT input table rows data structure

The leftmost part of the row (green, columns from 0 to 39) contains the values sampled from different oven sensors, measured at the same time. The central part (red, columns from 40 to 235) contains the mapping of the correspondent events. Information about events are available in Annex II (section 10.2.4.2.2). The rightmost part of the row (blue, columns from 236 to 240) contains the decibel values of the five acoustic sensors, registered at the same time of the corresponding sensors readings.

The network does not use the acoustic sensors, at the time of writing this deliverable, but they have been added to the structure for future use.

Operators’ feedbacks provided at M22 allowed refining the definition of failure, providing an insightful view of the procedure evaluation at the shop floor level, concerning the predictive maintenance scenario. The activity has been reflected directly into the dataset, with a comprehensive definition of states based on the following feature:

- hi-warning (columns 58 to 78);
- hi-deviation alarm (columns from 142 to 162);
- recipe loading (column 210).

Based on these features, the following rules have been adopted to define a fault:

1. A 30-minute stabilization period starts every time a recipe is loaded (.job file) and the oven starts the heating process. Hi warning events inside a stabilization period are normal. In the figure below three Hi warning (blue line) occur inside a stabilization period (orange line). This is not considered as a fault (the red line doesn’t have any step);

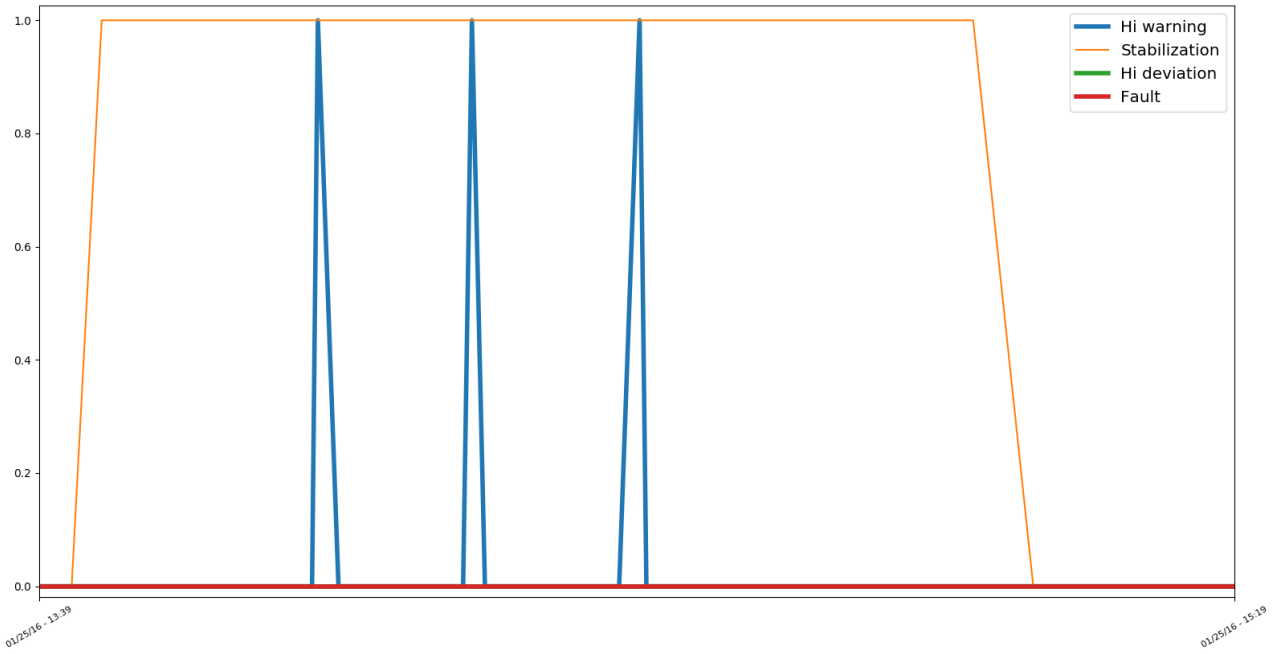

Figure 29: Hi warning in stabilization period

2. A cluster of hi-warnings after the stabilization period means that fans in that zone are not being able to stabilize the temperature. Therefore, after the stabilization period more than 3 “hi warnings” in a 30 minutes period, regardless whether they are consecutive or not, is considered a fault. Figure 30 shows a case where multiple Hi warning events occurred outside the stabilization zone. A fault is identified by a step in the red line.

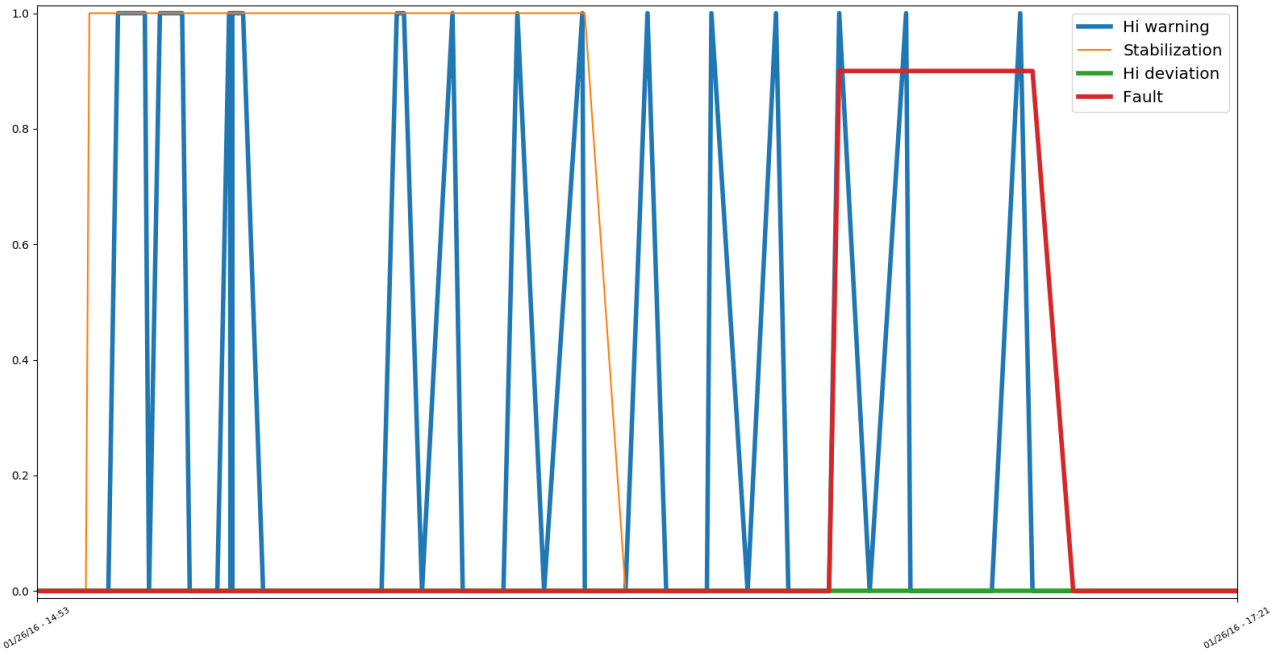

Figure 30: Hi warning outside the stabilization period

3. A hi-deviation alarm with only one Hi warning is sufficient to be considered a fault at any stage. In the figure below, a hi-warning and a hi-deviation (green line) occur sequentially. This is considered a fault also inside a stabilization zone.

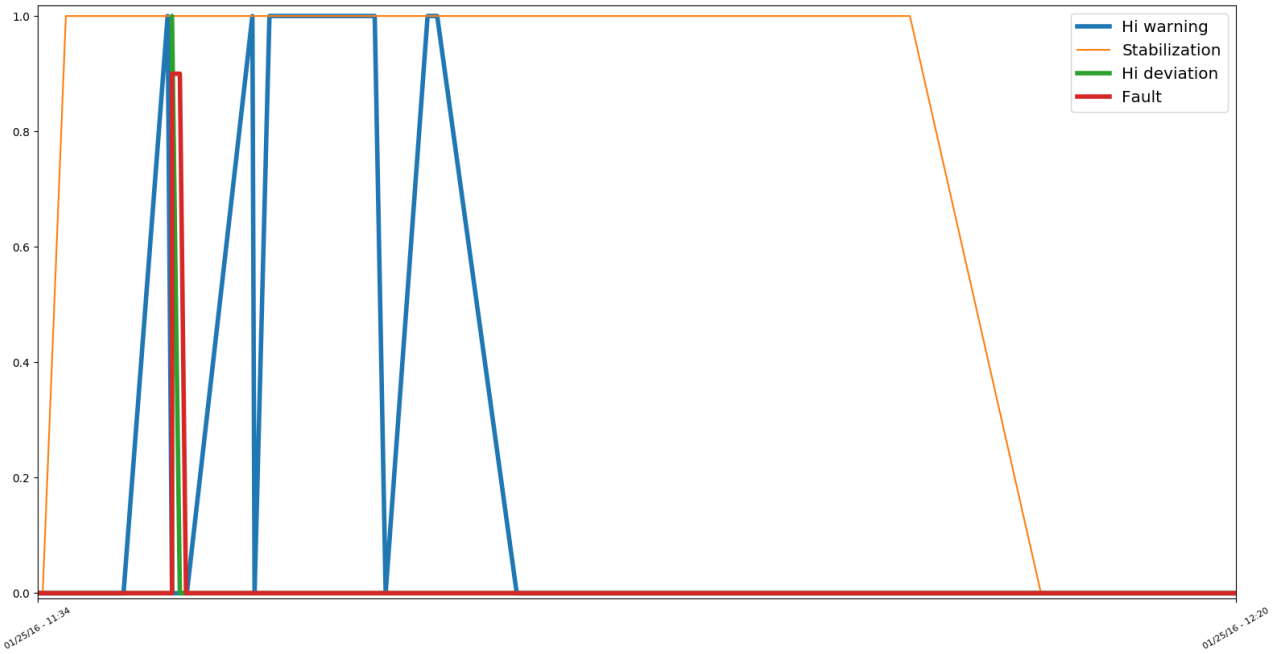

Figure 31: Hi warning and Hi deviation alarm inside a stabilization period

A total of 308 faults events have been identified in the historical period from 2012 to 2017, broken down as follows:

- 31 events with three hi-warning outside stabilization period (case 2)
- 271 events with an hi-deviation alarm with a hi-warning (case 3)

As for Brady oven, this is a case with imbalanced data where the classes are not represented equally: 308 failures over more than 378K total samples. In order to balance out input classes a weight has been assigned to each sample to correctly weight the loss function during the training phase. Furthermore, each of the remaining feature was normalized individually so that it is in the given range on the training set between zero and one.

The ANN model was trained using the same data structure described for Brady oven (section 10.2.4.1.1) with a different number of features. The feature containing the temporal distance from the last fault was removed because assessed as unfit by new tests. The resulting data structure is depicted in figure below:

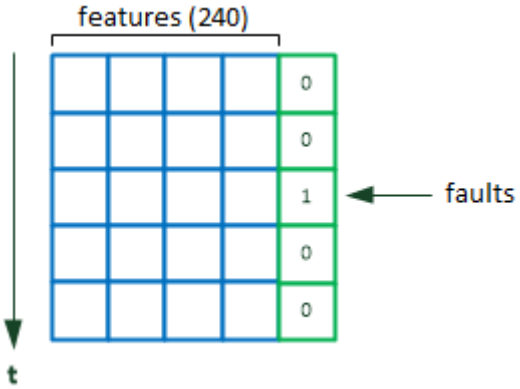

Figure 32: Oven features

The categorical cross entropy (Entropy, n.d.) loss has been chosen to suit the binary classification nature of this problem and evaluate the quality of the neural network during the training phase.

The network predicts the probability for each outcome mutually exclusive class as a value between 0 and 1. This represents the probability of a fault in the next 32 time steps (about 2.5 hours).

Initial tests have been undertaken on full extension of the input dataset (240 features) before proceeding to features selection methods to identify and filter redundant attributes from data that could harm the performance of the classification model. Then, less significant features have been discarded (e.g. the sensor communications), while sparse columns with similar information (e.g. warning in different oven zones) have been collapsed in a single feature maintaining the one-hot encoding pattern to improve the prediction.

The acoustic data could be extremely useful to extend the information gathered from the oven sensors, but, at the moment, the number of samples required to properly train an ANN is not reached. Preliminary analysis and experiments conducted on the existing acoustic data are described in section 6.1.2.2.

The resulting dataset is composed by 378K row each with 70 features. The figure below reports the training result on 40 epochs:

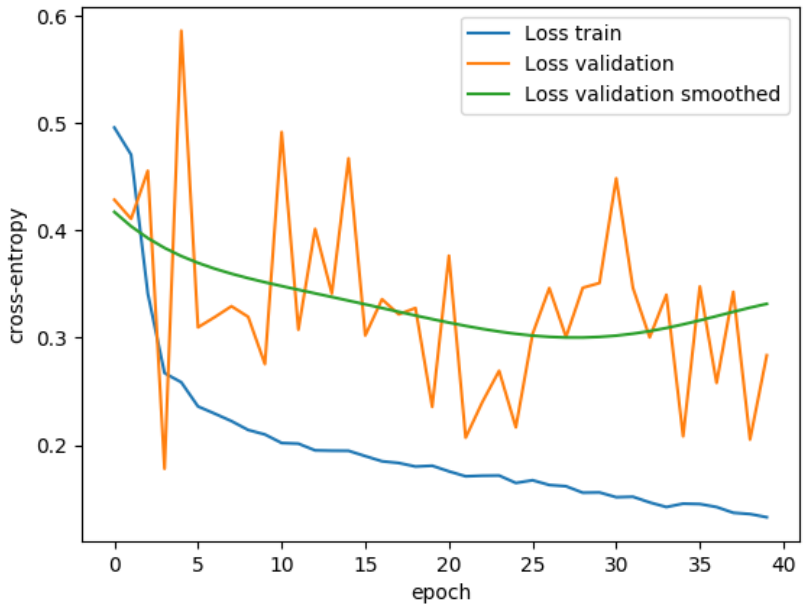

Figure 33: training results

In the figure above, it is trivial to see an inflection point around epoch 30. At this point, the slope of validation loss begins to increase while the train loss decrease continuously. Typically, this is the point where overfitting is beginning. The model at epoch 22 was chosen for the final independent assessment of the network generalization capability on the 2017/2018 dataset (86347 samples) that was never used in training and validation. Figure 34: ROC curve shows the receiver operating characteristic curve (ROC). It shows the diagnostic ability of a binary classifier system as its discrimination threshold is varied.

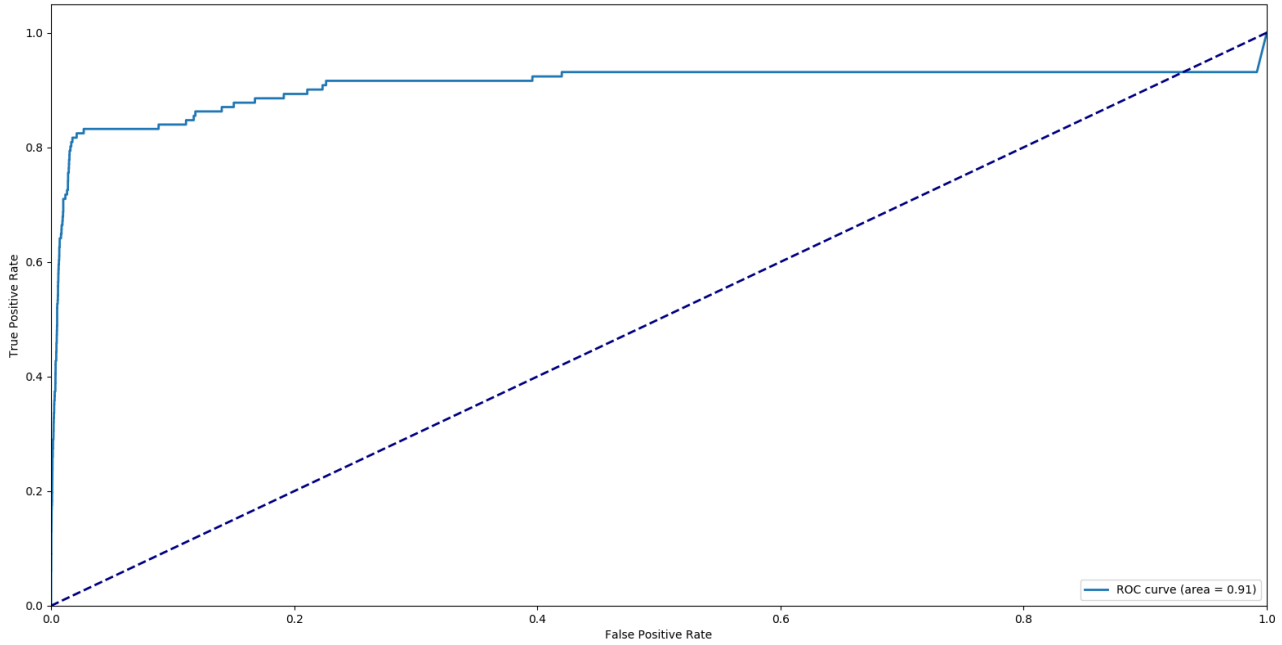

Figure 34: ROC curve

The optimal threshold should be chosen depending on the model goal. Confusion matrix in Figure 35 shows the performances of the classification model for the threshold of 0.84. This threshold was chosen to cover false positive rate (FPR) close to zero and true positive rate (TPR) close to one. More detail about confusion matrix are available in (Wikipedia, n.d.).

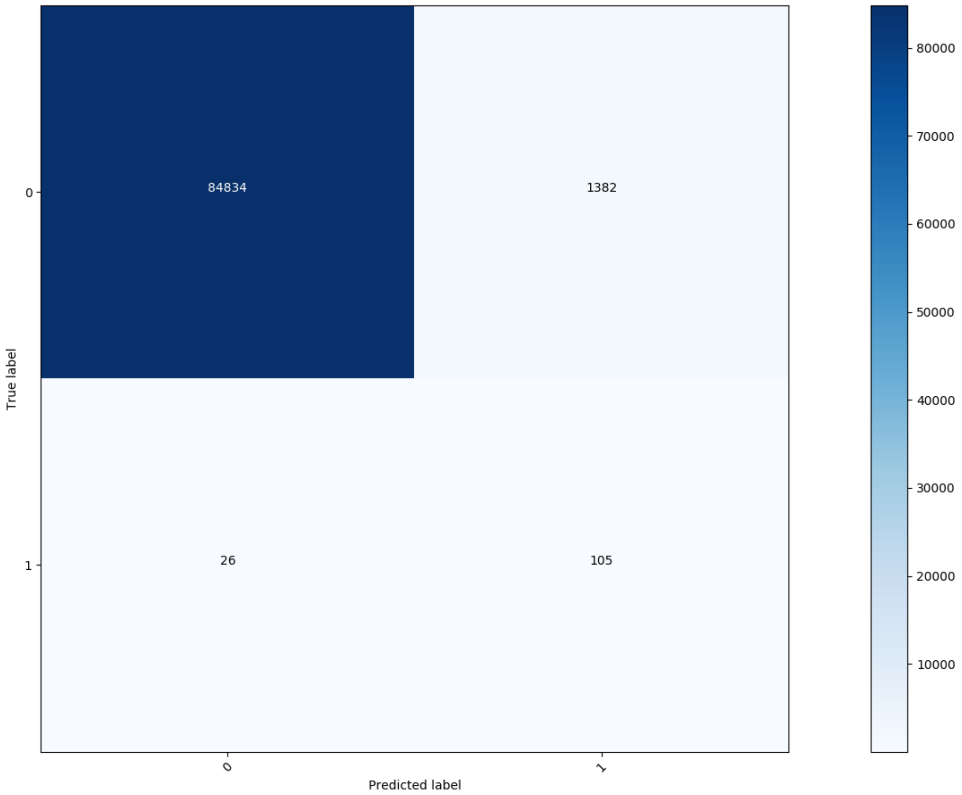

Figure 35: confusion matrix

Accuracy of the model can be calculated as:

$$ACC = \frac{(TP + FN)}{(TP + FP + FN + TN)}$$

In our case the accuracy is:

$$ACC = \frac{(105 + 84834)}{(105 + 26 + 84834 + 1382)} = \frac{84939}{86347} = 0.9837$$

It is worth noting that accuracy is not a reliable metric for the real performance of this classifier, because it yields misleading results for unbalanced dataset.

The normalized matrix in Figure 36 provides a better insight of the class accuracy with respect to both the predicted and actual marginal totals. Given that the row and marginal totals of normalized confusion matrices sum to a constant, respective cell values in two confusion matrices can be compared directly by inspection.

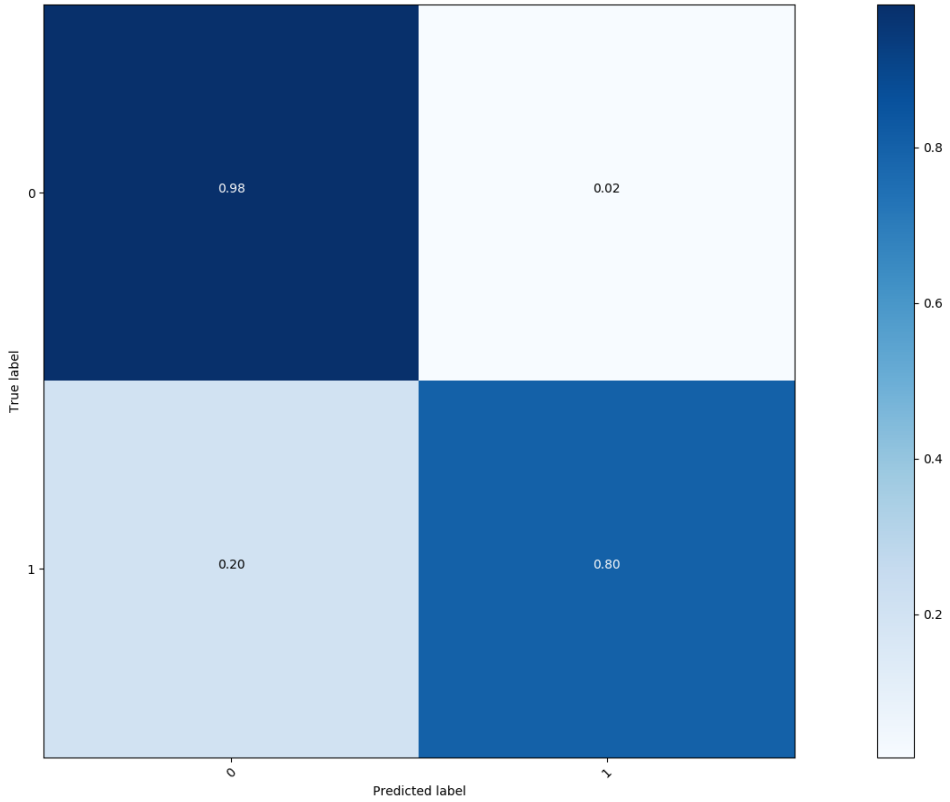

Figure 36: normalized confusion matrix

Figure 37 shows an overall view of the network prediction capabilities of the whole test data. The colours meaning is the following:

- brown plot are the hi-warning events;
- pink plot are the hi-deviation alarm events;
- purple plot define the stabilization zones;
- red points show when a recipe is loaded;
- orange plot represents the real fault;
- blue plot is the prediction of a fault.

The figure does not report all the available feature otherwise it would become too confusing.

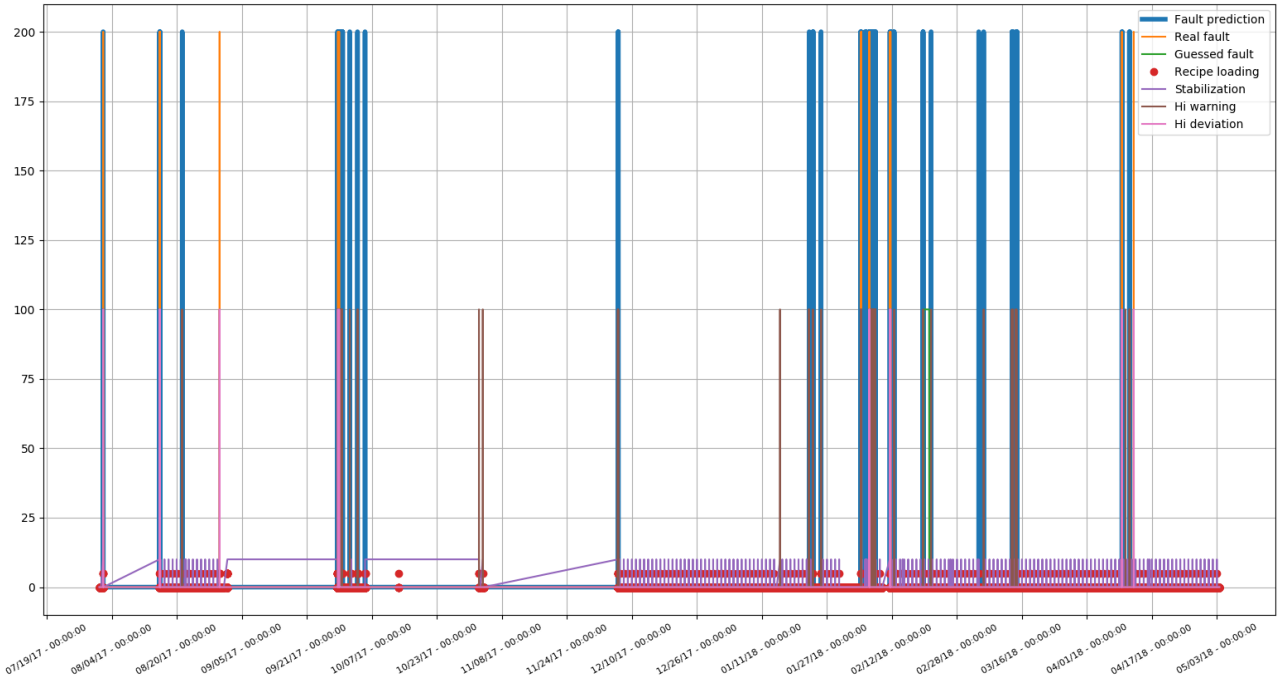

Figure 37: ANN prediction results on test data

It is possible to see some false positive. This happens due to the fact that the network is not completely able to ignore the fault pattern inside the stabilization zone. Simplifying the statements: the network understands the faulty patterns, but it does not understand that, since the oven is start upping, those patterns should be ignored. In this use case, an approach like this (that is more prone to false positives) is acceptable due to the nature of the prediction.

The easily visible “holes” inside the plot depend on the missing data inside the dataset, which come straight from the technique used for the creation of the dataset.

Since test data is distributed on a large timespan from the 19th July 2017 to the 5th March 2018, some zoomed-in details of the previous plot are showed in figure below. Figure 38 shows the network behaviour in the interval from 8:30 to 13:02 of the 6th February 2018. Some maintenance occurred starting from 31st January 2018 as reported by the following logs from BSL maintenance:

31/01:

- Rythmia oven put into maintenance due to conveyor limit adjustment

05/02:

- TC redundant alarms

05/02:

- Blower motor in zone 5 not starting

- Starting capacitor of motor in zone 5 replaced

- No heater elements turning on

- No heat contractor turning on

05/02:

- Over temperature detected in the bimetallic strip of the thermo couple

06/02

- Fan zones 5 and 6 not maintaining temperature

06/02

- Fans replaced in zones 5 and 6

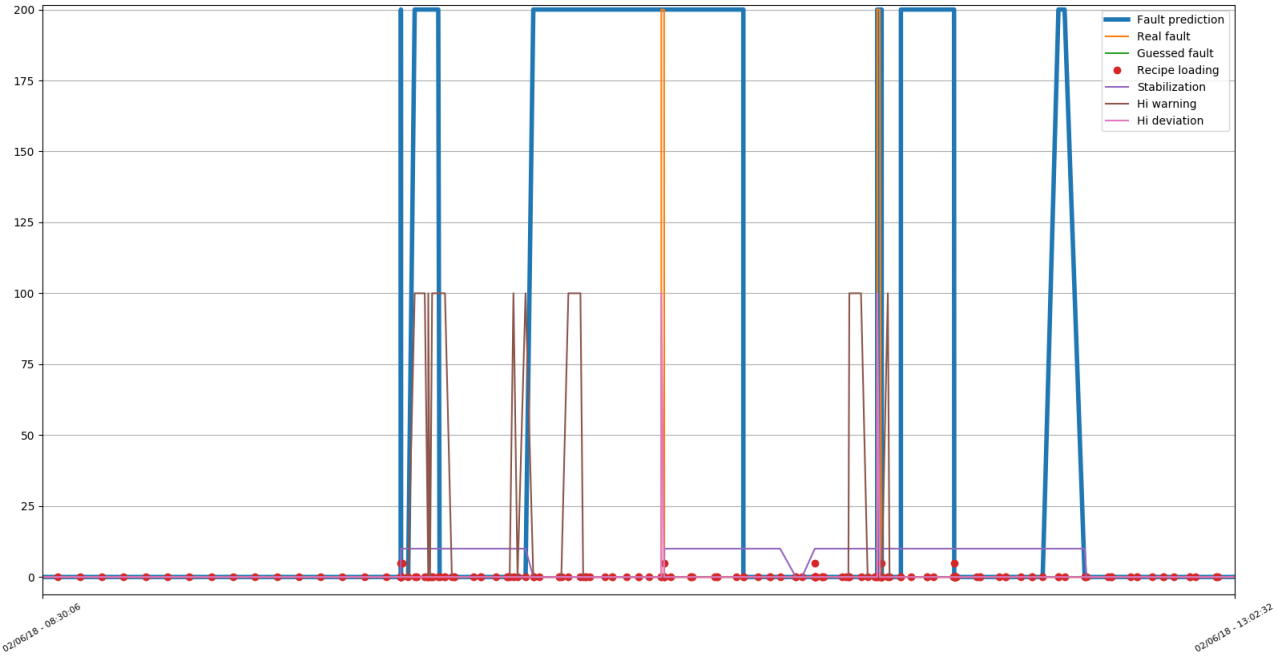

Figure 38: prediction result on real fault

The network is capable of successfully predict the fault (blue plot) of the oven before the pattern provided by BSL appears (orange peaks).

A positive result is also showed in Figure 39 that shows result in the interval from 18:30 of the 18th February 2018 to 20:15 of the 22nd February 2018. A faulty fan was fitted to the oven on the 19th February 2018 at 15:15 and was removed on the 20th February 2018 at 15:00 (green plot). The BSL pattern is not present in this interval, nonetheless the network can successfully predict the fault after the fan replacement but before the real detection.

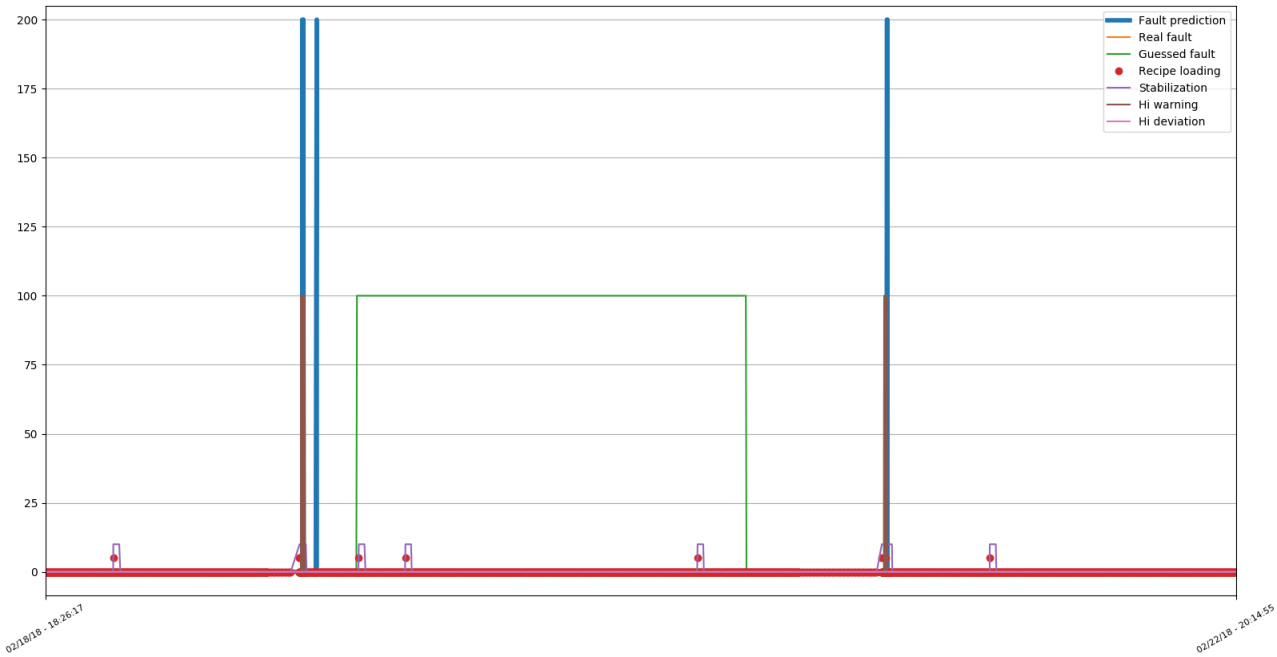

Figure 39: Test results

It is worth noting this detection, even if correct, negatively affects the metrics used to evaluate the quality of the network because it cannot be related to a real fault pattern. This highlights the extreme importance of correctly labelled data in order to successfully achieve the correct behaviour of ANN models.

In the right part of the figure, a false positive fault is also detected when the faulty fan is removed after the green plot step.

7.2.3 Audio data preliminary analysis and solutions

In Figure 40 we plot the data of the sensors in the time frame when they are all available. In this time window they are quite constant. The signal with higher variability is the one from PI2, while all the others are pretty stable, with PI1 which assumes slightly higher values. This is of course reflected also in their statistics: their variances range from 0.01005758 of PI3 to 0.06453504 of PI2 (6 times higher), while the means are between 98.6079811 and 97.88747826 for PI2, PI3 and PI4 and 100.04517391 for PI1.

Figure 40: Acoustic data

We can enlarge the window to include a period when not all the sensors are present, but most of them are. In this plot, it is clearly visible that the pattern of the sensors are very similar. The absolute values they assume are different, but their trend is basically the same. In particular, we can recognize two different values of frequency (or states) and all the sensors assume the low values in specific times and the high one in others.

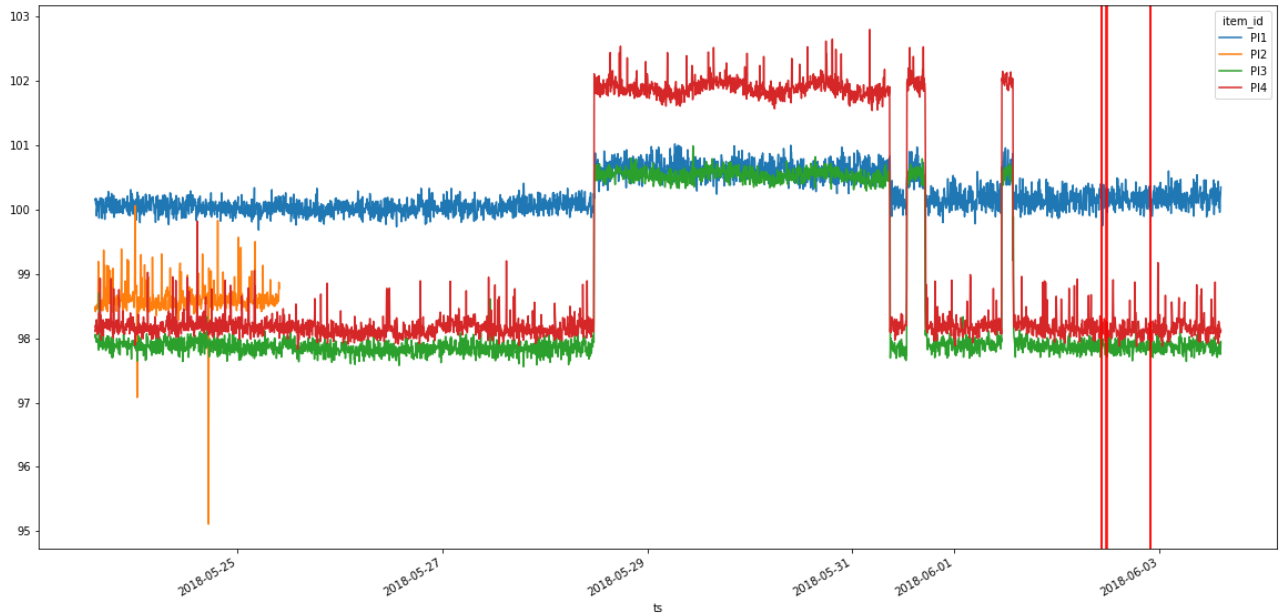

Figure 41: Sensor data 2

In the figure above, we inserted also vertical red lines when the oven logs reported a warning/failure condition (as detailed in 7.2.2). It is quite interesting to point out that there are some unexpected spikes on the data before the faults; nonetheless those spikes occur approximately 24 hours before the warning/fails, so it is hard to guess they are related. Anyway, more data is needed to draw any definite conclusion.

Indeed, the acoustic data provide a different view on the problem that can improve the quality of the proposed solution. Though the nature of the dataset is quite challenging since the data come from as set of tools "attached" to the oven that does not have the inherited synchronization between data and logs present on the Legacy data (6.1.2.1).

Writing this deliverable consists in the last step of the Task 5.2 "Continuous Deep Learning Toolkit for real time adaptation" and so forth there is no room left for the analysis and development of the new solution that will use the acoustic data. Despite this, the realization of a complete and even more stable DLT solution for COMPOSITION is central for us, thus more effort will be done in order to have a complete analysis and hopefully a solution which will be described in the deliverable of the WP8.

7.3 Comparison of CPU and GPU training

The section describes the comparison of CPU and GPU in a narrower setting of the UC-BLS-2. For comparison that is more generic, please refer to section 10.2.2.2.
The focus will be pointed on the training of a Recurrent Neural Network: first in a more generic fashion; then, in the second part, in the scope of the architecture explained in 7.2.1 and with data explained in 6.1.2.1.
The first comparison comes from the study of the different LSTM implementation of Tensorflow and Keras made by the author of the RETURNN framework (P. Doetsch, 2016)
There are multiple LSTM implementations/kernels available in Tensorflow. The aim is to compare the runtime performance during training for each of the kernels in a way as generic as possible
The following Keras kernels will be examined:

- Basic LSTM (CPU and GPU)
- Standard LSTM (CPU and GPU)
- LSTM Block (CPU and GPU)
- CuddnLSTM (GPU only)

The authors of the paper used the following configuration:

- Network: 5 layer bidirectional LSTM with dimension 500 in each time direction
- GPU: GeForce GTX 980

- CPU: 8 CPU threads

Table 10: Comparsion Kernel CPU - GPU

Kernel Type	CPU Time	GPU Time
BasicLSTM	05:00.5334	00:41.7282
StandardLSTM		04:57.7977
LSTMBlock	05:07.5613	00:33.4895
CudnnLSTM	----------	00:08.8151

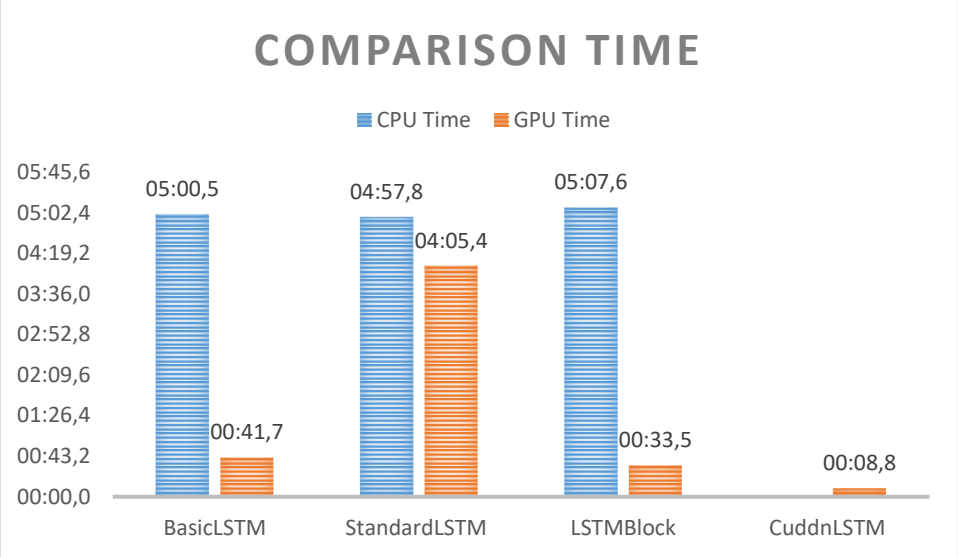

Figure 42: Graph of times different LSTM

For a more detailed comparison, please refer to: (RETURNN, n.d.). It is easy to notice in Figure 42 that the CPU time for the different kernels does not change that much between them; on the other hand, the GPU time of the CuddnLSTM is one order of magnitude smaller than the other GPU times. Thus, the best kernel in absolute is the CuddnLSTM, but since there is no implementation for CPU it is a choice that has to be taken carefully. If the need is different from the pure training speed, then the LSTMBlock seems to be the best one, offering a good speed and more portability since the implementation is present for both CPU and GPU. The second test is more specific to the needs of this project. In fact, it has been carried out using the network presented in section 7.2.1 with the data coming from the Brady oven (details here 6.1.2.1). The two kernels that were tested are:

- StandardLSTM
- CuddnLSTM

With the following hardware set-up:

- CPU Intel Core i7-7700K @ 4.20GHz
- RAM 64 GB
- GPU GeForce GTX 1060 6GB

The network and the data had the following parameters:

- Samples 137121
- Feature (32x72)
- Epoch 100
- Parameters to train: 136204

- (LSTM 128, LSTM 64, Dense 5)

- Batch 32

Table 11: Comparison GPU - CPU DLT network

Kernel Type	CPU Time (sec)	GPU Time (sec)
StandardLSTM		1:25:30.0
CudnnLSTM	----------	00:29:52.98

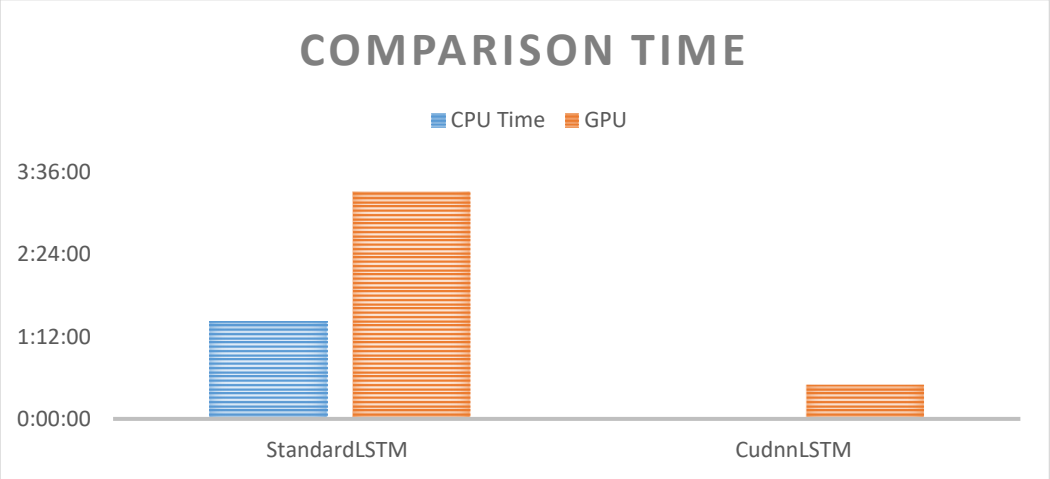

Figure 43: Graph of times DLT networks

The huge gap between the CPU and GPU time in the Figure 43 on the StandardLSTM comes from the overhead of copying the data from the RAM to the memory of the GPU. The overhead is always present for each GPU benchmark and it makes a big impact because - as shown in Figure 41 - the times of CPU and GPU are comparable. For this reason, since the dimension of the dataset is important, the copy takes longer and the CPU time is faster than the GPU time.

The conclusions that arise from the Figure 43 taken on the different kernels present in Keras: if there is certainty of the presence of the GPU on both the test and production server then the CudnnLSTM is the best choice. Otherwise, if it is necessary a more portable model, then the StandardLSTM is not a good choice compared to the BasicLSTM or LSTMBlock.

8 Deep Learning Toolkit deployment

During the first iteration of the component, information regarding each raw data stream from the shop floor (logs, sensors and acoustic data) were distributed through the Intra-Factory Management System. The Big Data Analytics module, now deployed as Learning Agent, is responsible for the operational procedures involving the aggregation and pre-processing of the aforementioned streams, in order to create a suitable mapping of the information to be interpreted by the Deep Learning Toolkit. The Deep Learning Toolkit returns previsions in reaction to incoming live samples, as expected. The architecture is described in the Annex I section 10.2.4.2.2.

In the second iteration, due to the peculiar nature of the exchanged data between the two components (rate, scope, usage, privacy, etc.), a one-to-one local communication strategy between the two modules has been introduced, in order to allow resources management optimization and logically separate local traffic from the intra-factory MQTT broker. In particular, the Big Data Analytics Complex-Event Machine Learning framework provides the communication functionalities needed to combine Complex-Event Processing and Machine Learning (ML) applied to the IoT as depicted in Figure 44.

Based on the above assumptions, the choice of creating a separate network with a private connection between the two containers has been made, linking them in a standalone manner. This means that the final configuration of the IIMF framework allows communications to the Deep Learning Toolkit Python backend through the Pyro remote method invocation functionalities. Pyro automatically serializes the data exchanged, reducing the effort and resources necessary for the OGC encapsulation. All the security issues derived by exposing a remote object interface through Pyro have been resolved through the trusted isolated network. Results and previsions from the Deep Learning Toolkit are provided only to the Learning Agent module that is therefore in charge to dispatch all the information to the designed containers autonomously.

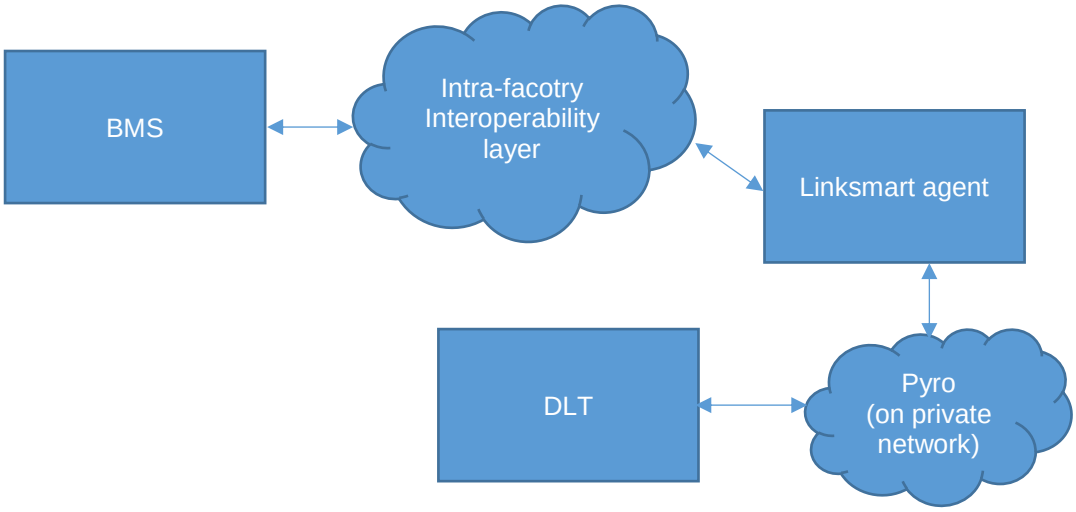

Figure 44: DLT for predictive maintenance

The DLT for predictive maintenance exports four different functionalities that can be remotely invoked by the LinkSmart processing agent through the Pyro framework:

- Predict: it generates output predictions for the input sample. It returns a Python list of three values:
  - prediction: 1 if the prediction is a fault, 0 otherwise;
  - accuracy: it is a decimal number between 0 (worst) and 1(best) that represent the accuracy of the prediction;
  - validity: the time of validity in minutes of the prediction. When a new prediction is generated, it overwrites all the oldest. For the deployed model this value is always 160 minutes.
- Predict on batch: the returning values would be [[prediction0, accuracy0, validity0], [prediction1, accuracy1, validity1], [prediction2, accuracy2, validity2]]

- Learn: this method can be used for continuous learning. It runs a single gradient update (unless specified differently) on a single sample of data.
- Learn on batch: this method can be used for continuous learning. It runs a stochastic gradient update on a batch of data.

The DLT also provides the methods importModel - which serializes and returns the model as a json object - and exportModel - which loads a serialized json model provided as parameter.

Whereas all the above is true for the intra-factory scenario, in the inter-factory a similar approach has been made for communicating with the Agents but with some small changes. The following image shows the changes from the intra-factory scenario:

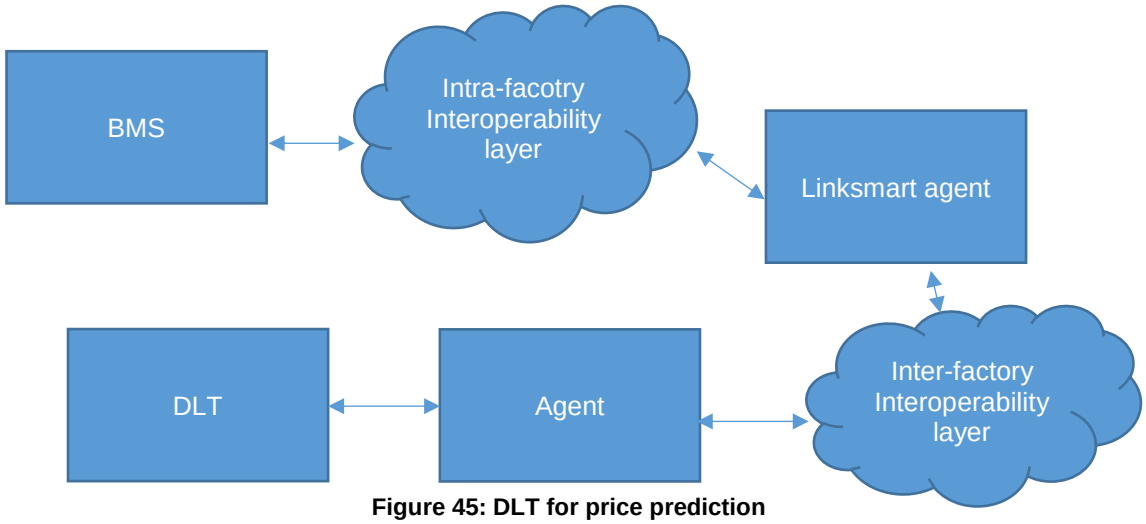

Figure 45: DLT for price prediction

Let us analyse the changes from the previous use-case. Now the DLT for price prediction does not work in the intra-factory environment but inter-factory. The single point of communication is with the Agent component and the DLT expose a REST API, using json as data protocol.

The so-called stakeholder agents are deployed at the stakeholder's premises and their purpose is to fulfil the stakeholder's interests. Two main kind of agents are foreseen in the COMPOSITION marketplace:

- The Requester Agent is the agent used by a factory to request the execution of an existing supply chain or to initiate a new supply chain. The Requester agent may act according to several negotiation protocols, which can possibly be supported by only a subset of the agents active on a specific marketplace instance.
- The Supplier agent is the counterpart of the Requester agent on the COMPOSITION Marketplace. It is usually adopted by actual suppliers to respond to supply requests coming from other stakeholders in the marketplace. Factories transforming goods typically employ at least one Requester agent to get prime goods and one Supplier agent to sell intermediate products to other factories.

The full description of the agents and the marketplace can be found in the deliverable "D6.3: COMPOSITION marketplace I".

As well as the previous use-case, a one-to-one local communication strategy between the two modules was introduced, in order to allow resources management optimization and a higher security on the communication between the two components. The secure communication has been obtained using a private network between the agent and the DLT. Due to the sensitive information that flows between agents and DLT this form of communication is needed.

The interface the DLT provides is a REST API. In details, the aim of the DLT has been expanded from a wrapper around a neural network framework for price prediction to a general-purpose system. It allows managing multiple goods, starting from not trained network and empowering the agent with the full control over the DLT and the inner ANN.

The main resource accessed through the API is the good. In the following table is provided the description for the available functionalities:

Table 12: Table of rest API of DLT

Path	HTTP Verb	Descscription
/good	POST	Creates a new good if it does not exist. It creates the correspondent network, untrained with the structure described in the section 7.1.1.
If the good exits, it adds values to it.		
/good	GET	Returns the list of all the goods created in the DLT.
/goods/{id}	GET	Returns the details about the good {id}.
/goods/{id}	DELETE	Deletes the good and its network.
/goods/{id}/values	POST	Adds a new value to the good. Each time a new value is added, the network predicts the new value, then the network is trained with the input value.
/goods/{id}/values	GET	Returns all the input values added through time.
/goods/{id}/predictions	GET	Returns all the predictions for a good in the form: [{ "value": 120.3, "date": 1524491482, "accuracy": 0 }] The value is the price predicted by the DLT. The date describes the time frame for which the value is valid. The accuracy is a value between 0 and 1 (1 best). It is the result of the r2 score calculated on all the previous predictions. The r2 is defined between -inf and 1 but in this case is truncated from 0 to 1 to cope with the previous defined interfaces.
/goods/{id}/predictions/last	GET	Returns the last prediction of the DLT (without adding a new value) with the same structure described above.

Indeed, when a new good is added, the network is untrained. Thus, its inner weights are random and it is necessary to train the network before using it. Theoretically, it will converge, but, as previously said (see 7.1.1.), it depends highly on the data that will be fed to the network.

The version of the DLT deployed on the production server already has 4 pre-trained models for the goods: paper, HDPE, PET, scrap metal. Those models can also be improved using continuous learning.

Taking in account the amount of hardware resources necessary for an artificial neural network, a maximum number of goods can be instantiated. After this limit, the following requests will fail.

9 Conclusions

The work that has been carried out in task 5.2 - Continuous Deep Learning Toolkit for Real-time Adaptation has been extensively described in this document. The activities have been divided in three consequential steps. The first step has been related to the creation of a suitable dataset for the two mainly addressed use case categories, related to both intra and inter-factory scenarios, ergo the predictive maintenance and the scrap metal price prediction. The second step has been related to more suitable research topics, such as providing a comprehensive analysis of the state-of-the-art, an in-depth overview of the available frameworks, and then followed by a detailed reporting of the extensive tests conducted. Both activities have converged in the first deployment of a first version of the Deep Learning Toolkit component in the lab scale project environment as a Docker contained image, for what concerns the predictive maintenance scenario. The third step is a sum of the first two, related to the creation of two different Deep Learning solutions for both use cases. These solutions are currently deployed in the COMPOSITION production server and are completely integrated in the work-flow with above described results and criticalities.

It is possible to include as main task achievements for the first two steps, the identifications of LSTM as the main sources of models for fitting the required scenarios. Also, Tensorflow have been chosen as the framework, thanks to its biggest support (maintained by Google and a strong open source community), to the fact that it provides the fastest training results, in terms of execution time, and thanks to the embedded support from the seamless usage of GPUs as computational power sources. Furthermore, on a more theoretical level, extensive tests on untrained LSTM Artificial Neural Networks topologies have provided the certainty to converge in a finite time span (under certain and strict conditions), whilst inputted with a periodic sinusoidal signal, in their untrained version, confirming the choice made.
It is crystal clear that the main achievement of this task was the ability to deploy a Deep Learning solution working on the Industry 4.0 domain, with production data coming from shop floor settings with limits and constraints defined by nature of the factory environment. Nevertheless, those sharp edges which the DLT has to deal with, it was able to provide relevant results, shown in the section 7.2.2. In the controlled environment, working offline on the test-dataset, the DLT has been able to make a step further for defeating “predictive maintenance chimera” anticipating a fault on the machinery before it was actually happening.

Following reviewer recommendations, it was analysed if the quality and quantity of the data were suitable for Deep Learning models. Regarding the use case UC-BSL-2 (predictive maintenance) with the amount of faults present in the dataset (only two), it was impossible to develop such solutions. However, after an effective exchange with the partners from BSL a pattern of event has been selected as a “faulty” with this approach the numbers of faults increased considerably (~2000) and hence it was possible to apply a Deep Learning Solution. Please refer to 6.1.3.1 to have a detailed description of the fault pattern.

Instead, regarding the use case UC-ELDIA-1 (price prediction) as already explained in 7.1.1 “if the boundary conditions when the network have been trained are the same of when they will be used to predict or to apply continuous learning the network will converge”. Otherwise instead if the price is affected by some features that are not present in the dataset (what is likely to occur) then the quality of prediction will degrade.

The work carried out in this task and in the whole work package will foresee major follow-ups in both applied research and innovation actions. The former will exploit further opportunities for applying artificial intelligence in the Industry 4.0 scenario, whereas the latter promises to have relevant impacts on different use cases.

10 Annex I

In this section of the document are reported relevant chapters and subchapter from the first iteration of this deliverable that has been submitted at M16, titled “D5.3 Continuous deep learning toolkit for real time adaptation I”.

Although this section is still useful and this information is still to be considered relevant, it has been moved to a separate part of the document allowing the reader to have a more comprehensive overview of the current state. In fact, by setting this information aside, the storytelling is enhanced and the process that brought to the final deployment gets to be more fluent.

Finally, all information below is not reported in a normalized manner, and their interdependence has to be considered negligible. They now form in this final version a track record for the process that is extensively highlighted in the main section of this document.

10.1 Unfitted use cases

The following use cases have been defined not suitable for a Deep Learning approach, due to their data nature or their expected outcome. Some of them have been approached in a statistical fashion by other tool of the composition ecosystem.

10.1.1 Maintenance Decision Support (UC-KLE-1)

10.1.1.1 Background

The Deep Learning Toolkit component is expected to distribute the latest prediction on the next expected failure of a BOSSI machine (used for metal surface finishing of pipes), based on the continuous input stream of sensors' data stream.

10.1.1.2 Data Overview

Kleemann provided a historical dataset of failures featuring:

- About 650 samples.
- About 20 features.
- Spanning 11 years, from 2007 to 2017.

10.1.1.3 Fitness for usage

Despite the consistent number of failures recorded (one per sample) and the congruous number of features over the entire time span, the dataset is unfit to be used on predictive maintenance and in general on every deep learning tasks, because of the following reasons:

- The dataset is just about failures but has no extensive collections of machine attached sensors where to look for patterns anticipating the failures.
- Data are sparse in time (opposed to sampled at constant frequency).

This data could probably be used with some success within a statistical data analytic framework, but this is out of scope for the Deep Learning Toolkit component.

10.1.1.4 Data assessment

Given the absence of a real dataset, DLT-based predictive maintenance in Kleemann seems not to be straightforward to implement. An untrained approach can be tried, inspired to the model setup that will be adopted for UC-BSL-2 after learning the related dataset. This might result in non-optimal model and will probably require long time to converge to an acceptable accuracy. Anyway, the precondition and also major criticality for this approach is the availability of live data streams from relevant machine attached sensors. At the present time (M16), such a sensors network does not exist in Kleemann, so the evaluation for applicability of the Deep Learning Toolkit in this use case is deferred to the next iteration of this document. Even if, sensors will be deployed and will provide meaningful live data streams, it is not granted that the recurrent neural network

that could be used in an untrained environment would converge to meaningful results, within the project timeframe.

10.1.2 Fill level classification use cases (UC-ELDIA-1 UC-ELDIA-2)

10.1.2.1 Background

The Deep Learning Toolkit component is expected to distribute the latest prediction on the fill level of waste material within a bin/container, in order to allow for optimization of timing and logistics of collections as well as improve any related commercial aspect. The prediction is based on a dataflow from one or more bin mounted sensors, monitoring its fill level.

10.1.2.2 Data Overview

Kleemann provided a minimal scrap metal dataset containing only 12 samples. They are equally distributed: one for each month of 2016. Each sample has 11 features, including the quantity of eight different metal scrap types and of 3 other materials (plastic, wood and paper) produced along one month. A consistent part of the data for forming a usable dataset is missing.

10.1.2.3 Fitness for usage

The data provided are currently unfit for the task of live prediction. The number of samples is several orders of magnitude too low and the time span is insufficient to detect long-term trends and seasonal patterns. Relevant data for this task would require one or more than one, time series of values acquired from bin-mounted sensors, related to its filling level, plus the collection events from the same container needs to be on record as well.

These data are not available, historically or live because both end users involved (Kleemann and Eldia) do not have any kind of sensors mounted on their bins nor their containers at the moment (M16). This action is planned to happen in the next months, so the evaluation of the applicability of the Deep Learning Toolkit in this use case is deferred to the next iteration of this document.

10.1.2.4 Data assessment

Given that no useful historical data are available, the only possible prediction task could be fulfilled leveraging on untrained Artificial Neural Network (or with Artificial Neural Network trained over a synthetic dataset). Nevertheless, although sensors will be deployed and will provide meaningful live data streams, it is not granted that the recurrent neural network that could be used in an untrained environment would converge within the project timeframe.

10.2 Deep Learning Toolkit preliminary design and testing

10.2.1 Introduction

As previously said for the sake of readability of the document the part which have been written in the first part of the project have not been deleted but have been moved in the annex. In particular this section in its completeness have been moved in the annex, but its content is still valid since describe the first steps of this related task.

This is the main chapter of this document and provides an in depth review to all the experiments that has been conducted from M5 up to M15. During this reporting period, Task 5.2 has put a lot of effort gathering data from end users, collaborating identifying suitable data sources matching project's use case. These data collection activity has been assessed in this document in chapter 6. In the meanwhile, extensive research on frameworks and technologies has been conducted, extensively described in chapter 5. On the top of all that, experiments on synthetic data has been conducted in order to identify technologies and testing different implementations of suitable artificial neural networks and corresponding algorithms in each of the frameworks. This activity has been conducted in parallel to the data analysis because creating realistic datasets is a huge time consuming activity. The aim is to analytically demonstrate that chosen elements of the artificial neural networks used to approach real world data, such as algorithms, activation functions, gradient descending iterative optimizations, balanced matrix of weights and so on, would converge to accurate results in a finite time span.

The first network topology investigated has been the Feed Forward Artificial Neural Network. It is a well-known approach in literature and widely adopted for solving classification challenges, like the one the use cases analysed can be broken down to. It has been investigated in all of the two very promising frameworks H2O and Tensor Flow. Both frameworks provide an implementation for it that is not very dissimilar from one another. The advantage of using Tensor Flow in this situation, resides in the GPU availability for incrementing speed scalar matrix operations, whereas H2O provides portable executable for easy testing and deployment. Moreover. Tensor Flow is the only one of the two that, thanks to third-party APIs allows a comprehensive implementation of Recurrent Neural Networks in all their topologies.

Tests continued with the two demos that has been presented to the consortium at M7 and at the first project review meeting at M9. The former in the form of a first demo, the latter using more accurate dataset in order to better mimicking the prices in the inter-factory scenario addressed.

In the meanwhile, some data that started flowing into the system and by becoming part of specifications, they took shape and therefore the need of advancing the network topology risen. In section 10.2.4.1.1 it is explained how time series has been shaped and modified to form the possible input for Recurrent Artificial Neural Network. The powerful regression that this ductile instrument could provide has been clear since the very beginning. The state-of-the-art analysis provided the result that the Long Short-Term Memory (LSTM) were the topology to look for in the combination of surveys and extensive results that taxonomy provides. At first, the focus has been put on the simplest and most used of this relatively new topology, the univariate variant. All experiments on synthetic data, real data based on the London metal exchange of aluminium prices and sinusoids trigonometric series are reported.

Despite the promising results provided by the LSTM univariate, the urge of adopting a more malleable topology arose when the data for the predictive maintenance scenario got tackled in. In fact, the number of features and the multidimensional model required by the input data for providing time step repetition over time and organizing incoming batches in a meaningful manner, has required the use of the multivariate version of this artificial neural network topology. In fact, the multivariate version has been used directly on real world data and the results are described in section 10.2.4.1.3. In specific, data from BLS and the Brady oven re-flower has been used for creating the first lab scale deployment of the Deep Learning Toolkit, leveraging on the LSTM Artificial Neural Network topology and models in its multivariate declination. Progressively better results have been achieved by improving the first attempt to use incoming data as-is by implementing clusterization of input data, and by imposing balanced classes constrains before feeding the Artificial Neural Network. Finally, the data normalization process has provided the last cog in the complex system in which the Deep Learning Toolkit design has resulted to be.

Finally, the chapter ends by describing the lab scale deployment in a Docker private containers that have been provided as software output of the task 5.2. The description concludes with four rounds of tests that has been performed on a periodical signal, in order to analytically prove the convergence of the model deployed in a finite time span.

10.2.2 Feed Forward NN in TensorFlow

10.2.2.1 TensorFlow introduction

TensorFlow is Google's own software framework for machine and deep learning. It is free, open source (Apache 2.0) and actively developed. It was created by the Google Brain team for internal use and first released on 9 November 2015.

Despite being a flexible and general purpose numerical computation tool, Tensor flow is mainly oriented to Neural Networks since its architecture is based on the computational graph paradigm: each algorithm is defined as a graph whose nodes represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.

TensorFlow is multiplatform: on 64-bit Linux, macOS, Windows, Android and iOS. Furthermore, it supports GPU computing via CUDA and since version 1.0 can run on multiple devices in parallel.

TensorFlow can also run on distributed clusters and process Big Data from Apache HDFS. It also features a flexible, high-performance serving system for machine-learned models designed for production environments.

The TensorFlow core is written in C++ for better performances, but for the sake of usability the main API is Python; C++, Java, Go are supported as well and many more unofficial binding exists including C#, Ruby and Scala. Furthermore, TensorFlow can be used as computational core from other high-level machine learning libraries including H2O and Keras.

The Python API has been used for COMPOSITION, with the option of adopting the C++ serving API for deploying in case the Python one should not perform fast enough. The API are layered in three different abstraction levels detailed in the following sections.

10.2.2.1.1 Low-level API

This is the base API: the most flexible and most expressive. The computational graph of the desired machine learning algorithm has to be created programmatically step by step, specifying each basic operation leading from the input tensor to the output tensor. To allow for evaluation and training nodes must be added to the graph implementing error metric computation and minimization.

Once the graph is finalized, it can be executed multiple times, with variable inputs. The typical use case for supervised learning is the following:

- Based on task to address and on the format of the dataset, choose model and hyperparameters:
    - NN type: topology.
    - NN size: number of layers, numbers of neurons per layer.
    - Optimization algorithm and training parameters (batch size, learning rate,).
- Implement the computational graph for the model and finalize it.
- Train on historical data by running the graph multiple times. For each training epoch:
    - For each batch in the training set:
        - Train the NN (execute the graph asking for the optimization node).
    - Evaluate the NN over a big batch of training data (execute the graph asking for the metric nodes).
    - Evaluate the NN over the validation dataset (execute the graph asking for the metric nodes).
- Plot training and validation metric trends:
    - To ensure the training process is sensible.
    - To detect criticalities in the model definition or in the dataset.
    - To assess if the training has converged or whether additional training epochs are needed.

The previous process is usually repeated multiple times with different hyper parameter values, and then the best trained model with best validation scores is retained and finally evaluated over a test set.

The resulting model can be deployed, used for prediction and periodically updated by incrementally training on batches of live data

10.2.2.1.2 Mid-level API

In this API the algorithm is still specified in terms of a computational graph, but its construction is eased by wrapper nodes representing different NN layer types. Each layer can enclose several operations: linear combination of inputs through weights and biases, activation function filtering, dropout, pooling, etc. So the layer API allow to write less code which is both more compact and more readable, at the price of a reduction of control over some details of internal nodes (such as weights and bias initialization).

Referring to python APIs, the official layer API is contained in the tf.contrib.layers package. Last TensorFlow version also natively integrated a popular machine learning API, named Keras. Keras is a mid-level python API too and is a convenient choice since it widely adopted even prior to TensorFlow so has solid documentation and a large community behind. Keras API in TensorFlow is located in the package tf.contrib.keras.

10.2.2.1.3 High-level API

TensorFlow also provide a higher level API, contained in the package tf.contrib.learn. This API totally give up the control over the computation graph and provide the data scientist with a restricted bunch of classes, each implementing a machine learning algorithm. Only a subset of algorithms is covered and advanced NN topologies are not available. Nevertheless, deep feed forward classifiers and regressors are featured.

These classes can be instantiated by providing the desired hyper parameter values and just work out of the box, providing methods such as train, evaluate and predict.

They both allow for quick tests with minimal coding effort and for usage without deep knowledge of computational graph mechanics. The major drawbacks are that there is no visibility on the finer details of the internal model.

10.2.2.2 Comparison of different TensorFlow API

Initially several supervised learning tests were conducted with TensorFlow to better understand it and assess its performances in different situations.

An open dataset dealing with industrial data was adopted at this stage. The dataset is related to a gas sensor array exposed to turbulent gas mixtures and is better described and fully available at (Gas dataset, n.d.).

For the purposes of this document, it is enough to know that the raw data were pre-processed so to obtain a dataset with ~3.45M samples and 11 features (timestamp, temperature, humidity and the reading of 8 gas sensors). For each sample, the target value was the Ethylene level, quantized in four classes (zero, low, medium and high). Time sequentially was broken with shuffling, and finally the samples were split into training, validation and test set (60%, 20%, and 20%). A first round of tests was conducted to compare the different Tensor Flow APIs so to get a gist of prediction accuracy, training time and to ensure that low, medium and high level APIs behave consistently to the finer grained low-level option. This can be considered a tie, performance wise, so the preferences is left to the implementation to perform technical decisions based on third party APIs features required and their compatibility.

10.2.2.3 Preliminary comparison of CPU and GPU training

Despite being extremely time consuming, training of Neural Networks is inherently highly parallelizable. This is why demanding the bulk of computing to multi core GPU is reported to speed up training up to a couple orders of magnitude.

Some preliminary tests about CPU vs GPU computing have been performed by training over the previously described gas dataset in order to assess the speed up magnitude.

The details of the two set-ups are described in 10.2.2.2.

Table 13: CPU vs GPU setup

CPU setup	GPU setup
Hardware	

• CPU: Intel Core i5-3570 @3.40GHz
  ○ # cores: 4
• RAM: 8GB
	• CPU: Dual Intel Xeon 2620 @2.10GHz
	  ○ # total cores: 12
	• RAM: 32GB
	• GPU: NVIDIA GeForce GTX 960 (AsusTeK)
	  ○ # CUDA cores: 1024
	• GPU RAM: 2GB

Software	
• OS: Windows 10 Pro • Python: 2.7 • TensorFlow: v0.8	• OS: Ubuntu 14.4 • Python: 2.7 • TensorFlow: v0.8, GPU enabled
Test Spec	
• task type: classification • model type: deep feed forward neural network • # input features: 11 • # output classes: 4 • hidden layers' size: [128, 64 ,32] neurons • optimizer: Adagrad • training batch size: 50 samples • # training batches: 2000	

When dealing with GPU computing of repetitive tasks, bottlenecks move from computation to data copying. Indeed, so to be available to GPU cores, the input data has to be copied from common RAM memory to the dedicated GPU memory. The same way, output data has to be copied back to the main memory to be logged or handled in any other way. Because of this, a special attention has to be paid to how the input data set is loaded from disk and fed to the training process. Tensor Flow offer various mechanisms for data loading. The most general one is the Dataset API suitable to build complex input pipelines and to deal with huge datasets that cannot fit entirely in the memory. These datasets usually come as large collections of binary (or less frequently textual) files, which may reside on multiple hard disks, either locals or NAS, potentially abstracted by a distributed file system layer such as Apache HDFS. Given the manageable size of the adopted dataset that entirely fits in memory this mechanism is disproportionate: simpler data acquisition is desirable. This may be implemented in different ways also depending on the chosen API as discussed in the next sections.

10.2.2.3.1 Low and mid-level API

Two types of approach to data loading are available:

- Placeholder: in the tensor flow graph, tf.placeholder objects, whose value can be fed at runtime, represent the batch of input samples and the related targets. With this approach, the dataset is loaded at once as a plain python numpy.ndarray. At each training step, a minibatch is sliced by the ndarray and internally converted in the tensor form to replace the placeholders.
- Tensor: in this approach the whole dataset, samples and targets, is loaded to tf.tensor objects since the beginning. Other tensors are defined to represent the batch. This can be more time efficient since no format conversion is required at run time but it has several practical disadvantages:
  - The batch size has to be predetermined so that the significance of validation metrics is severely compromised because of the limited number of samples. Alternatively, it is possible to define a conditional graph, which depending on run time flags can behave differently (e.g.

training, validation and test). Anyway, this makes the code mode complex and bug prone, less readable and reusable.

- Additional operations must be added to the graph to bridge the dataset tensors and the batch tensors, resulting in reduced flexibility. In particular, two choices are available:
  - Filling the batch with consecutive slices of the dataset.
  - Filling the batch with picking random samples from the dataset.

  The first way requires minimal computational overhead, while the second make easier to have different batches across different training epochs, potentially leading to a better accuracy.

Reports results of tests performed with the three described data feeding strategies, each performed with three different hardware setups: the CPU and GPU configuration described before plus a second CPU setup running on Dual Intel Xeon 2620 @2.10GHz (12 total cores) with 32GB of memory.

Table 14: Low-level API - CPU vs GPU

Data loading mode	CPU [ms/batch]	Dual CPU [ms/batch]	GPU [ms/batch]
numpy.ndarray	65	68	67
tf.tensor, random pick	0.49	0.94	1.8
tf.tensor, slicing	0.47	0.86	1.45

Unexpected considerations emerge from these data:

- The placeholder plus numpy.ndarray approach is on average two orders of magnitude slower when compared to tf.tensor data loading, meaning that the on-the-fly conversion of batches from Numpy (S. van der Walt, 2011) to TensorFlow data format is very taxing or prevent some optimized behaviour from taking place. In this kind of approach, the time cost of data conversion prevails so that almost no differences between different hardware setups can be identified.
- The tf.tensor-based data feeding lead to significant speedup, the slicing version being slightly faster that the random picking one.
- The first unexpected outcome is that comparing CPU executions, the consumer targeted Intel Core i5 with 4 cores and 8GB of memory performs twice better than a couple of server meant Intel Xeon 2620 with 12 cores and 32GB of memory.
- The second unexpected outcome is that comparing CPU and GPU executions, the latter performs up to 4 times slower than the former, suggesting a severe bottleneck due to copying the dataset chunks from the CPU memory to the GPU one, which can hardly affect the training time.

Further investigations will be probably performed in the next iterations of this document in order to understand if further software releases of used APIs and frameworks will address the GPU compatibility in a more consistent manner or if this is dependent by the used hardware that is not capable to exploit latest compilation options and drivers features.

10.2.2.3.2 High level API

For the high level API, the data loading alternatives are quite similar to the previous case, but there are less flexibility drawbacks are using the tensor approach. In the tests, we compared three different loading approaches:

- The dataset is loaded as numpy.ndarray objects and passed to the fit method of the Neural Network object.
- The dataset is loaded as tf.tensor objects, and a function is passed to the fit method of the Neural Network object, which compose the next batch by random sample picking.

- The dataset is loaded as tf.tensor objects, and a function is passed to the fit method of the Neural Network object, which compose the next batch by consecutive slicing of the dataset.

The results of the test are detailed in Table 15: training time are reported per batch and expressed in milliseconds. The validation has been disabled during the tests not to distort the time measurements. External times are computed as the total training time divided by the number of training steps and directly compares to the one reported is the previous round of tests, while the internal times are the average of per step times as obtained through call-backs provided by the Neural Network object.

Table 15: High-level API - CPU vs GPU

Data loading mode	CPU [ms/batch]		GPU [ms/batch]	
	external	internal	external	internal
numpy.ndarray	2.3	0.8	11	9.5
tf.tensor, random pick	9.1	0.7	12	3
tf.tensor, slicing	10.4	0.6	11.7	2.6

These are surprising and unexpected results: in the given setup, CPU computing performs systematically faster than GPU. Furthermore, in the CPU tests, the Neural Network object train significantly slower if fed with tf.tensor batches, which should be the faster approach not requiring additional conversion from numpy formats.

In CPU tests there are major discrepancies between time measured internally and externally to the training session, probably meaning that internal measurements do not include overhead tasks such as saving model checkpoints and above all assembling the batch to process from the dataset; this could explain why the internal time is mostly constant while external time can increase up to 5 times between different data loading approaches.

Externally measured GPU time is very similar internal one when the dataset is provided as numpy.ndarray. Contrary, when the dataset is provided as tf.tensor objects the internal batch time is greatly reduced while the external one does not vary significantly.

There are different hypothesis that could explain the weird CPU better than GPU results:

- TensorFlow version 0.8 might have buggy implementation of GPU computing, leading to unnecessary slowdowns. It would be interesting to perform a new set of tests with the most recent version.
- There might have been issues in the test workstation or in the GPU environment setup even is the installation procedure reproduced step by step the official instructions and even if execution logs did not let any issue or criticality emerge.
- Most likely there is a major bottleneck due to copying the dataset chunks from the CPU memory to the GPU one, which can hardly impact the training time, independently from the data structure adopted (numpy.ndarray vs tf.tensor)

Finally, by globally comparing training times on CPU using high level API with those obtained with low level API, it can be seen that low level API train about one order of magnitude faster when using tf.tensor dataset and about one order of magnitude slower when using numpy.ndarray dataset. So when training time become an issue, either because the datasets are large or because multiple experiments with different hyperparameters (e.g. grid search) are to be done, it seems better to use CPU computing, with low level Tensor Flow API and fed the dataset as slices from tf.tensor objects. As a matter of fact, the high level APIs provided by Keras had provided the best and more ductile approach for the problem classes that the COMPOSITION use cases required.

Despite these not so promising results, the GPU training will be further investigated in the next iteration of this document because it is common practice and well agreed among insiders that is the way to go. Problems may reside in the version of the APIs used or in some missing compilation flag of the correspondent modules.

10.2.3 Feed Forward NN in H_2O

Given the partially inconsistent outcomes from TensorFlow in the tests previously described, it seemed desirable to try and compare a different framework in order to check is similar issues persists and if TensorFlow result are reproducible with a different framework or if better results can be achieved. In order to provide consistency across comparisons the same problem class is addressed.

H_2O was chosen as secondary framework. Experiments are detailed in the following sections.

10.2.3.1 H_2O introduction

H_2O is a multiplatform java-based open source toolkit for machine learning and big-data analysis. Its API is high level and extremely user friendly featuring a web-based GUI and bindings for Python, R and Scala. It can target other frameworks as computational core, including TensorFlow, but its native java core is extremely fast and scalable, being able to handle large datasets.

The plethora of supported ML algorithms is wide but, concerning deep neural networks, only feed forward classifier/regressors and auto encoders are supported.

Since trained models can be exported as plain java classes, it is easy and fast to integrate and deploy them in any java pipeline.

While GPU computing is not straightforward, distributed clusters are natively supported and H_2O seamlessly integrates with cloud computing technologies such as Apache Hadoop Distributed File System and commercial services such as Amazon Web Services.

10.2.3.2 Regression tests

A number of experiments with increasing complexity were carried out. In depth review of each of them would be of limited significance. Nevertheless, it is worth to briefly describe them to report the progression from simpler tests to more complex use cases that led to the first demo presented at M7 review meeting and discussed in depth in the next section. All tests are concerned with the supervised regression task over synthetic data.

- Test H_2O 01: the dataset is generated from univariate exponentially decreasing trend, corrupted with additive uniform noise.
- Test H_2O 02: same as Test H_2O 01 but grid search approach has been used to extensively explore and select hyperparameters best values.
- Test H_2O 03: similar to H_2O 01 but the dataset is multivariate: additional random features are added to the significant one in order to increase the difficulty of training.
- Test H_2O 04: same as Test H_2O 03 but grid search approach has been used to extensively explore and select hyperparameters best values.
- Test H_2O 05: the univariate dataset is generated from a trend that combines an exponential decay and a sinusoid and corrupted with additive uniform noise.
- Test H_2O 06: given the problems in the previous test, here the dataset is generated from a single univariate sinusoid trend, corrupted with additive uniform noise.
- Test H_2O 07: similar to H_2O 01, but the dataset is multivariate by assuming the independent variable to be a time measure and decomposing it into 3 features: year, month and day.
- Test H_2O 08: similar to previous test, but comparing results obtained by training over dataset of different dimensions (varying the sampling frequency).
- Test H_2O 09: two different exponential trends are mixed in the same dataset and a second feature flags has been added to each sample, specifying the trend it belongs to. The regressor must learn the two different trends and to discriminate between them based on the additional feature.

10.2.3.3 First prices prediction demo

At M7 it has been demonstrated to the consortium the first draft component for predicting prices within a range in a controlled environment. This demonstration has been performed on synthetic data and without using continuous learning techniques.

The sequence of test previously described culminated in the development of a synthetic price regression demo based on a deep feed forward network trained on a synthetic dataset. This demo was presented to COMPOSITION partners at M7 project meeting.

The addressed use case was UC-KLE-3 (Based on D2.1 v0.7), concerned with determining price for scrap metal. In this context the DLT can provide previsions of price fluctuation per scrap type per commercial partner.

In particular, Eldia provides price offer to Kleemann for exact tonnage of scrap and determines the price based on quotes from its customers, aiming to optimal scrap reselling with minimal storage. Forecasting of quote prices from various customers can support Eldia agent in timely decision making:

- Asking for less, but more relevant quotes
- Adopting estimated quotes instead of real ones in case of lack of time
- Providing past interpolated and future estimated price trajectories

The simulated historical dataset of quotes gives a full knowledge of the ground truth generating function allowing for better performance evaluation.

The dataset spans over 70 years from 1950 to 2019, each sample is a quotation for a specific scrap type (out of 4) and from a specific customer (out of 4). Also each quotation is relative to a specific quantity of scrap, ranging from 1 to 10 tons, which nonlinearly impacts the target price.

The dataset contains 48 different trends, each contributing about 300 samples and resulting is a total size of 14400 samples corresponding to 16 quotations per month. The datasets have been split respecting the proportion of 80% for training, 20% for validation and 20% for testing.

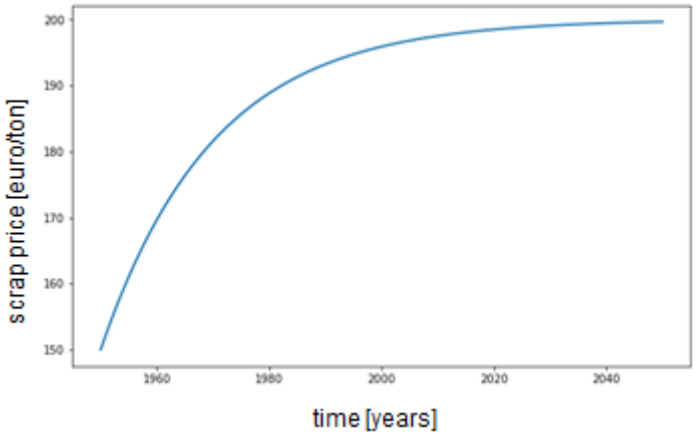

Figure 46: ground truth generating function of a single price trend

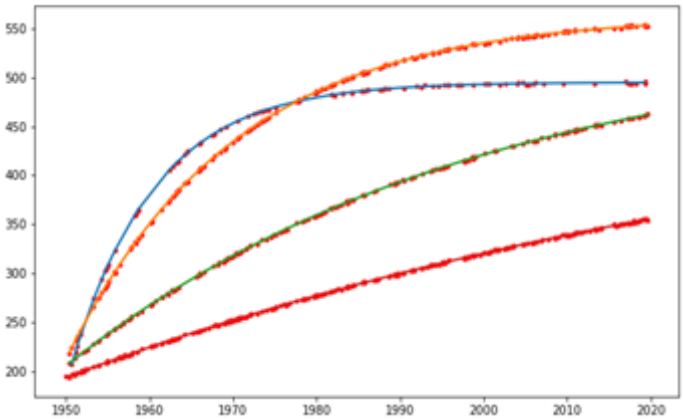

Figure 47: four different price trends: ground truth function and noise corrupted samples

Each sample composed of eight features, plus the target price, namely: year, month, day, scrap type, customer id, scrap quantity, scrap colour and quote time to live.

The last two features are uncorrelated to the target price and are intended as a disturbance co complicate the regression task.

The regression network is a feed forward deep multi-layer perceptron (MLP). A satisfactory arrangement concerning hyperparameters have been determined through a grid search over them, leading to the following choices:

- 4 hidden layers
- Neurons per layer: [8, 128, 64, 32, 16, 1]
- Activation function: f(x) = tanh(x)
- Automatic metric selection for error evaluation
- No regularization

The adopted training specification are:

- Training epochs: 30
- Minibatch size: 5 samples
- Learning rate: 0.1
- Adaptive learning algorithm: adadelta
- Data normalization: enabled

The monitoring was carried out at each epoch with training batch size of 4000 samples and validation batch size of about 2000 samples.

The scoring history shows a significant reduction of overall error, as shown in Figure 48, and a continuous convergence of training and validation error.

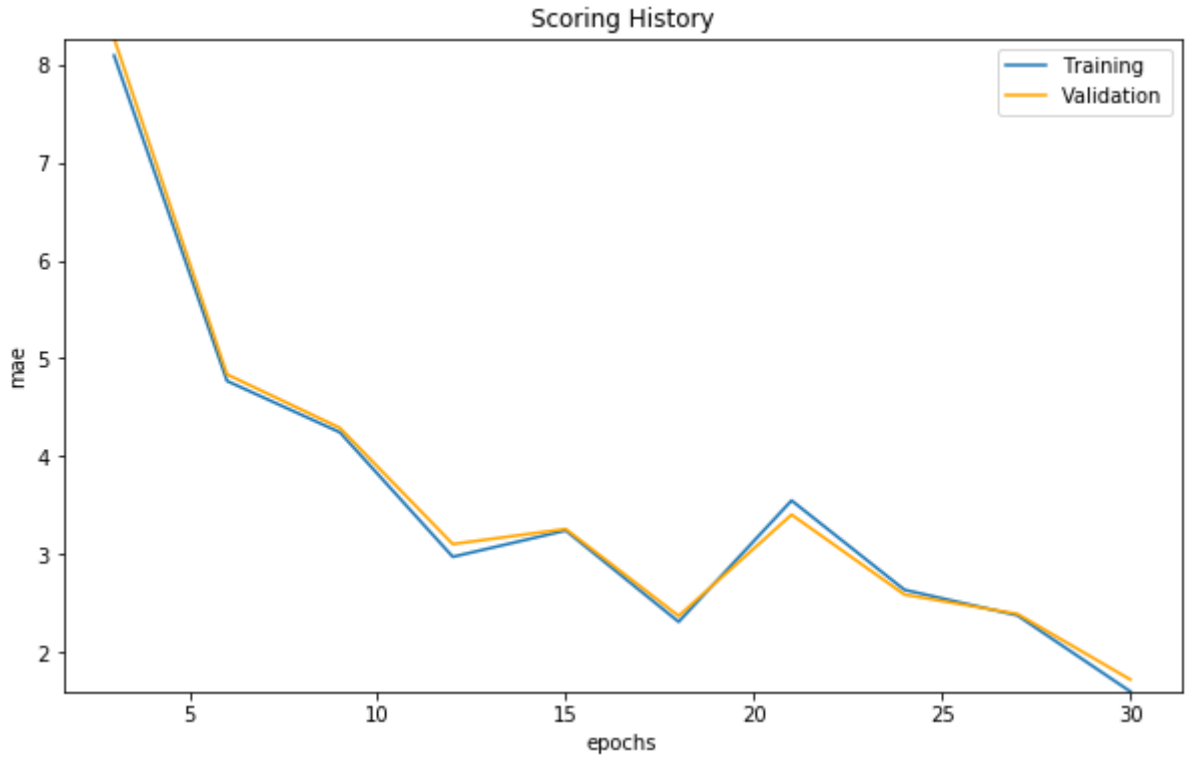

Figure 48: variation of mean absolute error (MAE) along training epochs

A number of final performance metrics are reported in Table 16 for the train, validation and test data. As expected the test errors are wider that validation and training, but the increase is very slight to demonstrate the good generalization capability of this model.

Table 16: final performances of the trained neural network

Error metric	Training set	Validation set	Test set
Mean Squared Error (MSE)	8.871	8.836	10.034
Root Mean Squared Error (RMSE)	2.978	2.972	3.167
Mean Absolute Error (MAE)	1.979	2.0315	2.018
Root Mean Squared Logarithmic Error (RMSLE)	0.00914	0.00887	0.00959

It is worth considering that the theoretical minimal MAE is bound to 0.5€ by the additive noise and that the obtained results are higher but comparable with this value, while they stay two orders of magnitude below than the average price along the dataset which is about 250€ (so that the average relative error is about 1%).

The good overall performance of the network in discriminating between the 48 trends and make sensible predictions for each is well depicted in Figure that shows for a subset of the trends, the ground truth function and the predictions made by the network, the two curves are in good agreement.

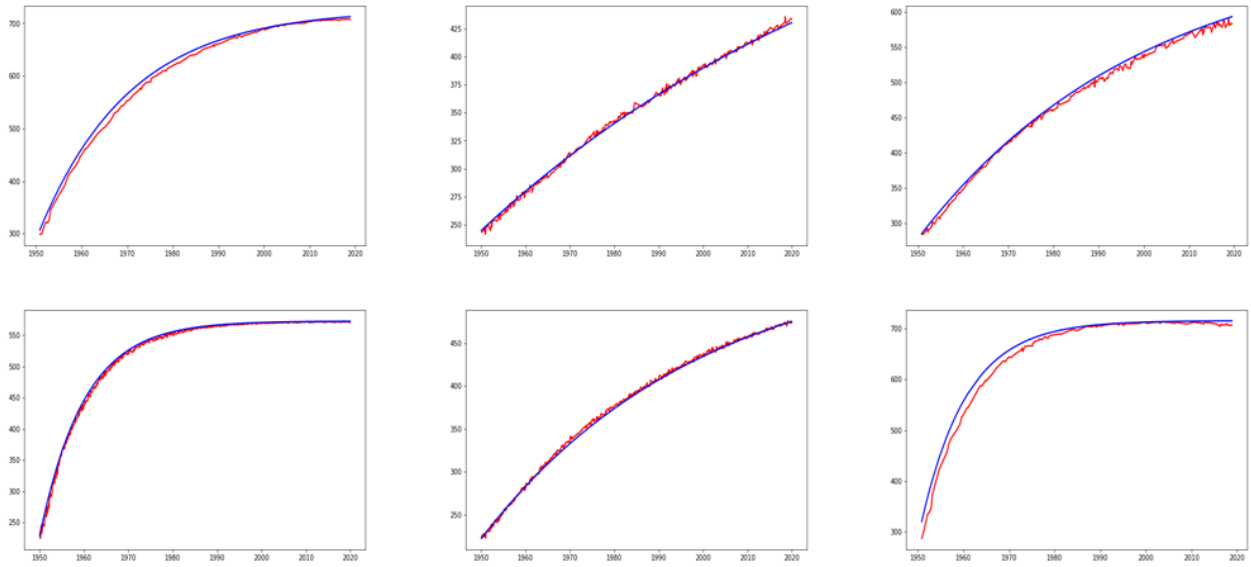

Figure 49: Different plot of the function

10.2.3.4 Updated demonstration on prices prediction

The first demonstration described in section 10.2.3.3 was updated and demonstrated at the review meeting at M9. The significant novelties were two. First and foremost, the testing process was extended to determine the ability of the network to make predictions in the future, beyond the end of the historical data set. Three smaller test set were created from the same ground truth generating functions, each spanning over a one-year period: namely the 2020, 2021 and 2024. Testing on these datasets accounted for network performances at 1, 2 and 5 year predictions.

The expected error increase is mitigated by the asymptotic structure of the price trends in the dataset and appear to be noticeable but acceptably limited as reported in Table 17.

Table 17: predictions' evaluation of future datasets

Error metric	2020 dataset	2021 dataset	2024 dataset
Mean Squared Error (MSE)	7.139	8.332	16.331
Root Mean Squared Error (RMSE)	2.671	2.886	4.041
Mean Absolute Error (MAE)	1.945	2.106	2.921
Root Mean Squared Logarithmic Error (RMSLE)	0.00513	0.00567	0.00804

The second major step up was the deploy of the demo as a standalone docker image wrapping the python script files in a python enabled virtual debian environment. The image is designed to be run locally because the demo works as a console application, and thus need access to the host windowing system (X11 under linux, Xming under Windows). The dockerization of the demo has been accomplished and allow to replicate it under both Linux and Window platform without need for specific dependencies (python, various python modules including H_2O, java) which are all wrapped into the docker container.

Even if intended for local use the image has been hosted on the official docker Portainer instance of COMPOSITION.

10.2.4 Recurrent NN for time series regression

The experiments detailed up to this point showed that Feed Forward Neural Networks are capable to effectively fit algebraic functions such as exponential, even when corrupted with various amount of noise, and deliver sensible predictions even outside the range of the training set. Furthermore, these networks are able to learn a trend from multivariate input even if the correlation with the target is distributed among several features, and they effectively learn to ignore features that share no correlation with the target (e.g. random features).

In addition, Feed Forward Neural Networks can learn datasets that contains several different trends, discriminated by one or more categorical features, and predict using the correct trends based on the values of the current samples.

We also discovered that few samples are required to learn a single exponential trend (300 are enough) even in presence of consistent amount of additive noise. In spite of that, several concerns about these network emerged. In particular, they do not seem able to handle periodic functions in a convenient way, with severe errors achieved immediately beyond the range of input domain covered by the dataset. This is a major issue for regression tasks dealing time series either univariate, such as in prediction of bin fill level, or multivariate, such as in predictive maintenance.

Recurrent Neural Network (RNN) are a much more convenient family of Neural Network for dealing with time series. Their topology in based on neuron-wise cyclic graph, resulting in each neuron having an internal status that can keep track of previously analysed samples to process and predict the current one. This intrinsic ability to handle the past history, is the most beneficial and attractive feature of RNN: to mimic the same kind of behaviour with a conventional Feed Forward network, the number of required parameters would be way greater also requiring wider datasets to train on and/or more training epochs.

RNN are a group including several different kinds of networks. Among RNN, the state of the art is the topology called Long Short Term Memory (LSTM) networks. Unlike most simple RNN which can keep track of a predefined number of previous time steps, LSTM neuron has a complex internal structure in which a number of gates control the information flow. This allows the neuron to selectively decide at each time step which part of the internal state has to be forgotten and which part of the current input is used to update it.

Thus LSTM has theoretically and unlimited back time horizon; in practice it is bound by computational resources dedicated to training and, of course, by the dimension of the training set which is finite.

In conclusion, given the recent state of the art achievement of LSTM networks in a plethora of different fields ranging from Natural Language Processing (NLP) to Computer Vision (CV), and given their predisposition to handle time series, they have been chosen to fulfil the DLT multipurpose nature.

In switching from Feed Forward to LSTM networks, also we abandoned H_2O, that does not support RNN to TensorFlow and in particular to its embedded Keras API. Keras, created in 2015 by François Chollet from Google, is a pure python library for neural networks, meant to be an API running interchangeably on top of an external core among MXNet, Deeplearning4j, TensorFlow, CNTK or Theano. It is simple to use, widely adopted by a constantly growing community so that Tensor Flow decided to incorporate it.

Keras was developed aiming at four principles:

- Modularity in neural network definition.
- Extensibility to include new type of neural network layers as they are invented.
- Minimalism in the API to keep the code simple, compact and readable.
- Native python, no custom file formats for neural network specification.

Keras is built around the concept of Sequential model which is a generic neural network, whose topology is specified as a stack of layers appended one after another. All the relevant layer types are available to create models, including LSTM layers. After the model is complete, Keras requires it to be finalized through a compile command. After this step, it can be trained through the fit function and finally evaluated or used for predictions.

10.2.4.1 LSTM regression on univariate time series

10.2.4.1.1 From time series to dataset

In order apply LSTM network to make predictions over a univariate time series, the first mandatory operation is the pre-processing of the dataset in order to reshape it into a proper format.

Some definitions are provided in the followings:

- The original time series is composed of measurements of a single feature (since it is univariate) sampled uniformly along the timeline. These scalars are referred as time steps of the series. Let's suppose the data available to fit the network counts up to a number of total time steps, named n_ts_all.
- On the opposite side, the dataset we want to obtain is composed of a couple of matrices: samples X and targets Y. They have the same number of rows and number of samples (n_samples): each sample has an associated target; when the network is fed with a sample, its target represents the corresponding desired output. The i-th sample is a row vector denoted as xi. The i-th target is a row vector denoted as yi.
- Each sample xi is composed of a fixed number of timesteps, named n_ts_in, representing how far back the network can look in order to deliver a prediction. The larger n_ts_in, the more information available to the network but the more taxing the training.
- Each target yi is composed of a fixed number of time steps, named n_ts_out, representing the temporal horizon of each prediction. The larger n_ts_out, the more challenging is to deliver good predictions.

Figure 50 shows in a self-explaining way how to convert a time series into a dataset. Consecutive samples are obtained by shifting by one single time step along the timeline the biggest possible dataset. The total number of samples can be calculated as follows:

$$n\_samples = n\_ts\_all - n\_ts\_in - n\_ts\_out$$

At this stage, values n_ts_in and n_ts_out have to be chosen based on the specific task to address. An elevated n_ts_in does not only impact the complexity of training, requiring more computational time per training step and exponentially more training epochs, but also it determines a reduction of n_samples, that can get percentually considerable as n_ts_in get closer to n_samples.

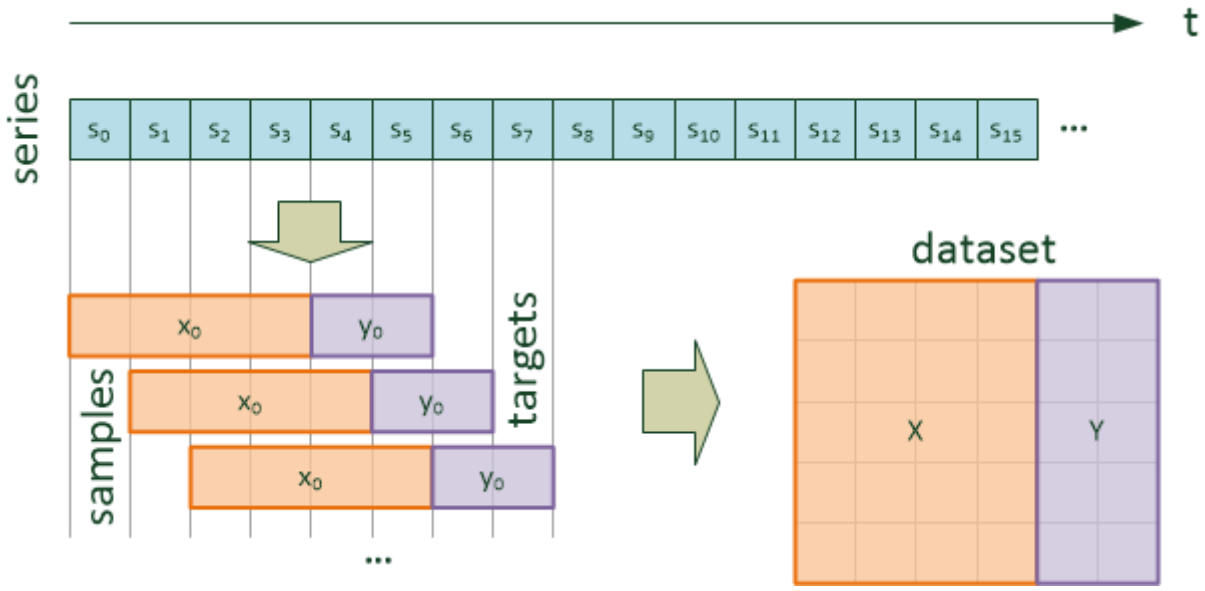

Figure 50: construction of a dataset from a time series (n_ts_is = 4, n_ts_out=2)

Also in the process of creating the dataset from the raw time series, manipulation of the target can be performed aiming at simplifying the training task. Indeed, time series prediction is a challenging task, so that it is common practice in real applications to make it as simple as possible.

For example, let's consider a scenario, where the future price trend of some goods has to be predicted based on its past fluctuations. A way to simplified version of the task is to try and to predict, instead of the future price, whether it will be higher or lower than the current value (or than the average price of the last n_ts_in timesteps). In this way, the prediction task is simplified from the regression of a continuous variable to a binary classification problem. This is simple to learn because only two target class are to be learnt from the samples in the dataset.

Another technique that can be adopted to simplify the problem is compressing the values to predict. With respect to the previous example, if the price is sampled once a day, in order to make a one-week prediction n_ts_out should be equal to 7, so that the network outputs seven values, predicting one price for each of the following days. Seven guess are more complicated to hit than one. The task can be simplified predicting only a scalar output, being the average price along the next week, or directly the price at the 7th day from now.

Since the neural network learn what to predict based on the targets of the training set, the way to implement the relaxation of the learning task is the pre-processing of the targets by quantization or aggregation.

10.2.4.1.2 Dataset shuffling and splitting

Once the dataset is ready it can be shuffled along the vertical dimension, that is altering the order of the samples (and correspondingly of the targets) in X and Y matrices. The reason to perform such an operation is to ensure a random distribution of targets. For example, if the dataset is concerned with predictive maintenance and targets are binary broken/not-broken values, let us suppose that a significant majority of breakages happened is the second half of the time series. If the dataset was unshuffled it would present a non-uniform distribution of breakage targets this impact the training. Since this is typically performed iterating many times over the entire training set, let us consider the last training epoch, leading to the final network. Being and adaptive process of network weights tuning, recent training steps are more impactful that the older ones. If all breakage targets come at the end of the dataset, in the final training steps the network will be presented with more failures than the average, incurring in the risk it gets to prefer predicting these events more often than needed.

Since shuffling only affects samples order but leave both the sample-target correspondence and the sequence of time steps within each sample unchanged, this will not impact the capability of LSTM network to learn from temporally correlated data.

Another necessary operation is to split the dataset into three portions dedicated respectively to training (usually 60% of samples), validation (20%) and testing (20%) as shown in Figure 51. The indicated percentages are of common use but are not mandatory. If a final independent assessment of the network generalization capability

is not the most relevant point and the n_samples is tight, the testing partition can be skipped and data divided 70-30 or 80-20 between training and validation. Also for the sake of performance evaluation with small datasets, more complicated cross-validation schema can be adopted.

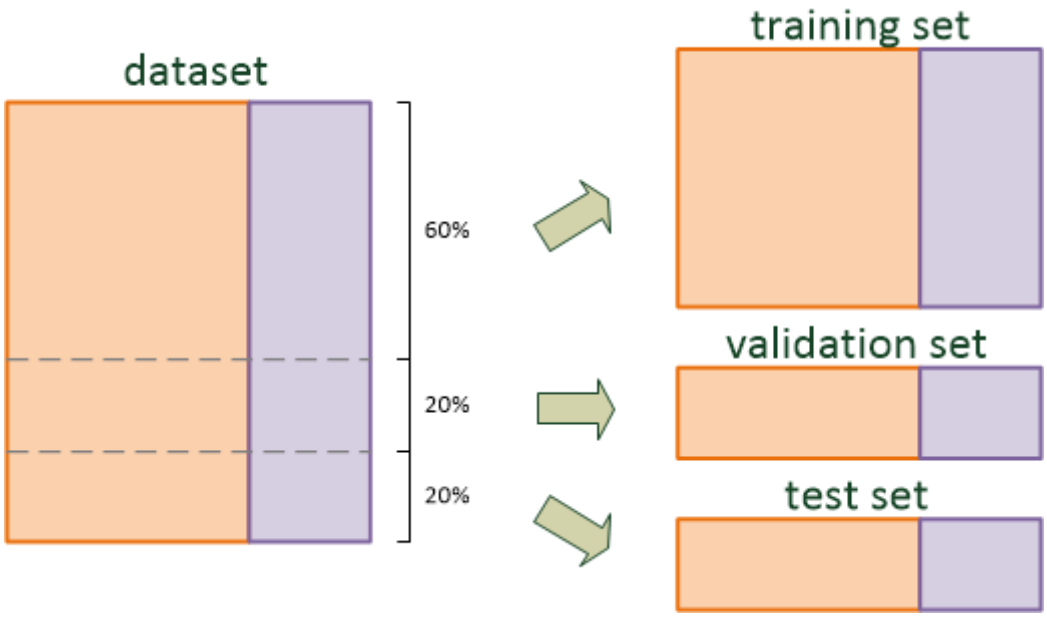

Figure 51: Dataset splitting

10.2.4.1.3 LSTM network design, training and evaluation

In the figure below (Figure 52) it is showed the overall resulting architecture that has been adopted for the models used in the ANNs and specifically realized during the experimental processes.

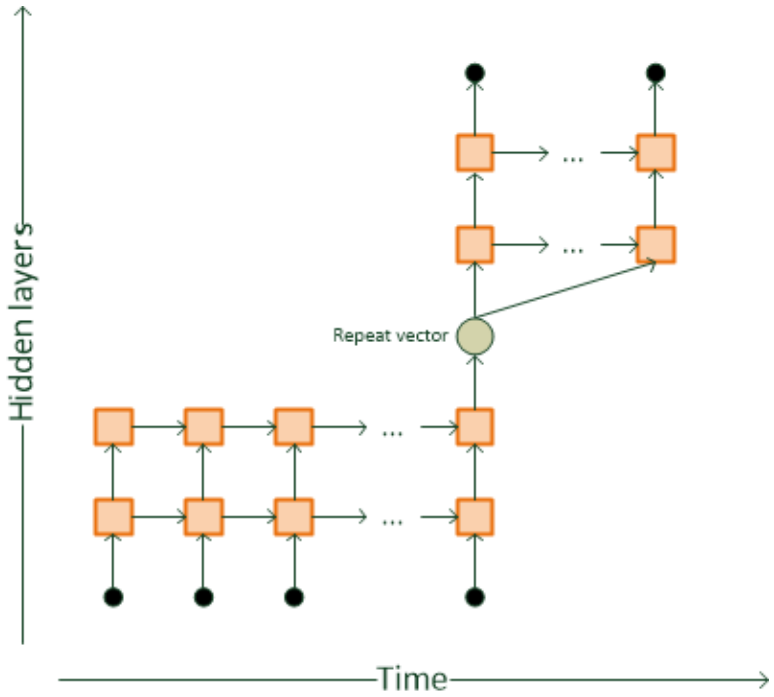

Figure 52: LSTM model architecture

The model consists of two different groups of LSTM layers, spaced out by a Repeat Vector element between the two. The second group has the function of repeating the initial time series by N-times, in order to match the output vectors number of timestamps. Therefore, each layer can have a different number of neurons. On

a normal regime, lower values have a lower impact on the final prediction accuracy, whether higher values provide better accuracy, despite having a negative impact on training speed in terms of completion time.

The hidden layer hyper-parameters are not fixed, but they are heavily dependent by the experiment type. It is worth mentioning that the same model will be used for all the experiments described in the sections of this document. In particular, for univariate time series two experiments, detailed in the sections 10.2.4.1.3.1 and 10.2.4.1.3.2, have been realized:

- A metal price estimation within fixed prevision timeframe
- A trigonometric dataset to test the correctness of the proposed network model design.

10.2.4.1.3.1 Metal price estimation

As reported in the data assessment is section 6, the scrap metal dataset by Eldia has not reached, at the moment of this report is released, the number of samples required to properly train an ANN. In order to fulfil this vacancy, a public database of price trends, available from London Metal Exchange (London metal dataset, n.d.), have been considered to provide a much larger and more comprehensive data set. INERISCI FIG reports the selected interval of 1400 price samples from 2012 to 2017.

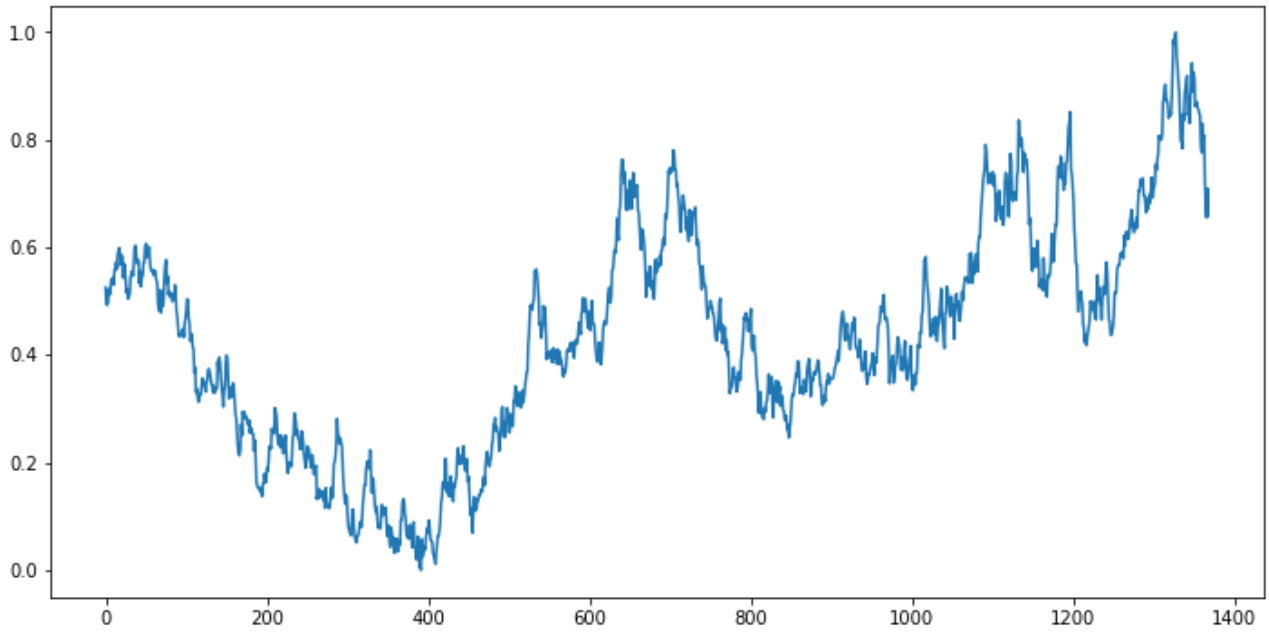

Figure 53: London Metal Exchange aluminium prices

The raw values are converted into samples with:

- n_ts_in = 60 timesteps
- n_ts_out = 15 timesteps

The initially untrained model is composed by five hidden layers with the following neurons deepness: 16, 16, 16, 8, 8. In Figure 54 are reported the results of the training process on 905 samples. The remaining samples are used to validate the predictions.

The plot on training data (blue) converge rapidly after few epochs. The validation plot (orange) follows the same trend but with a quite low precision.

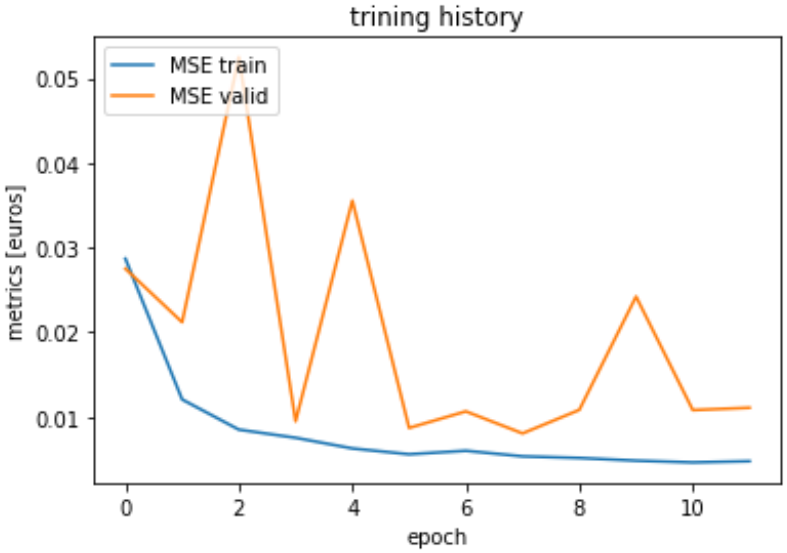

Figure 54: Training result

The network can predict 15 time steps in the future with increasing predictions errors. In the Figure 55 and Figure 56 three plots are displayed:

- The blue plot: the immediate prediction of the next value; lowest error.
- The orange plot: the most distant prediction in time; highest error.
- The green plot: represents the original dataset.

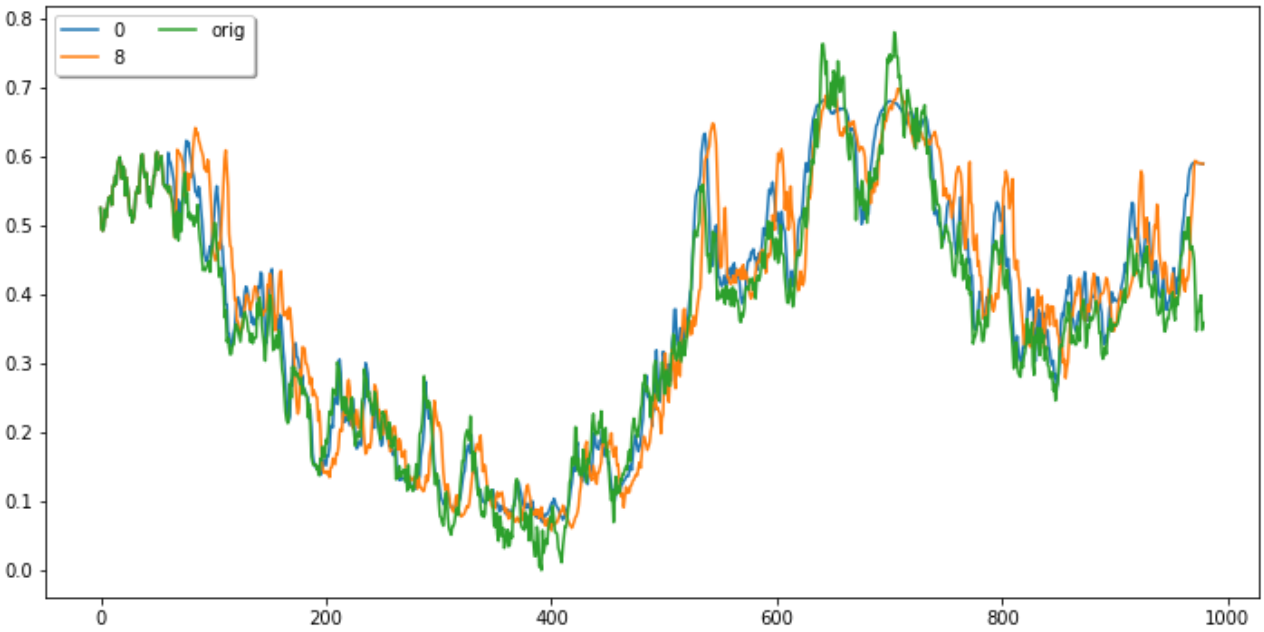

Figure 55: prediction results on training set

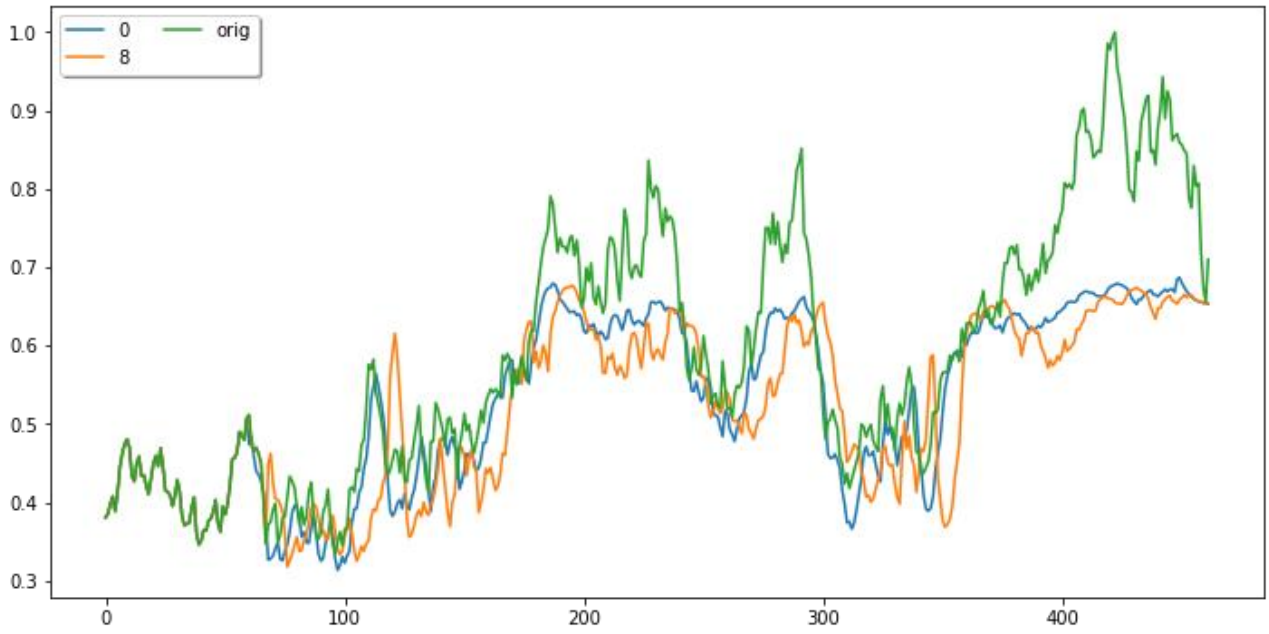

Figure 56: Prediction result on validation set

As expected from the training results, the predictions' quality on the validation set (Figure 57) is less precise than the one produced with the verification of training set (Figure 55). A higher precision and therefore higher quality can be achieved by increasing the number of input samples.

10.2.4.1.3.2 Trigonometric series

In this experiment, a total of 4000 raw values have been obtained from the trigonometric function depicted in Figure 57.

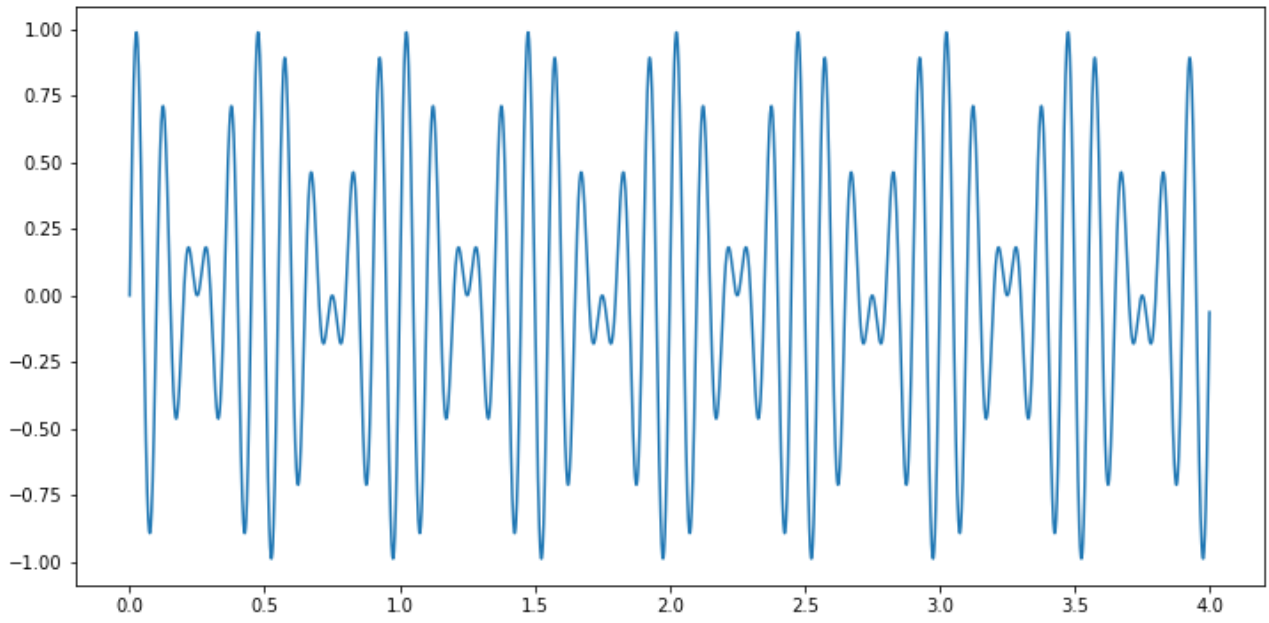

Figure 57: Input time series

The raw values are converted into samples with:

- n_ts_in = 90 timesteps
- n_ts_out = 30 timesteps

This results in a total of 3880 samples.

The initial network for regression is composed of four hidden LSTM layers of size 16, 16, 8 and 8 neurons respectively, with hyperbolic tangent as activation function. The network has been then trained on 2716 samples and subsequently validated on 1164 samples. Figure 58 shows the training result.

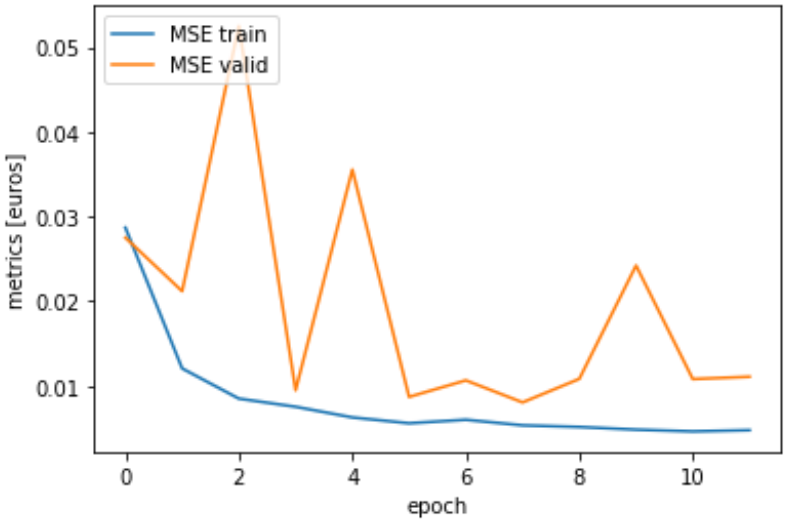

Figure 58: Metrics

It is easy to notice how in this case, again, as well has it has been described in the previous experiment, the network is able to predict up to 30 time steps in the future with a trend of decreasing accuracy in predictions' precision. In the Figure 59, four plots are displayed:

- the blue plot: the immediate prediction of the next value; lowest error;
- the green plot: the most distant prediction in time; highest error;
- the yellow plot: in between of the previous two plots;
- the red plot: represents the original dataset.

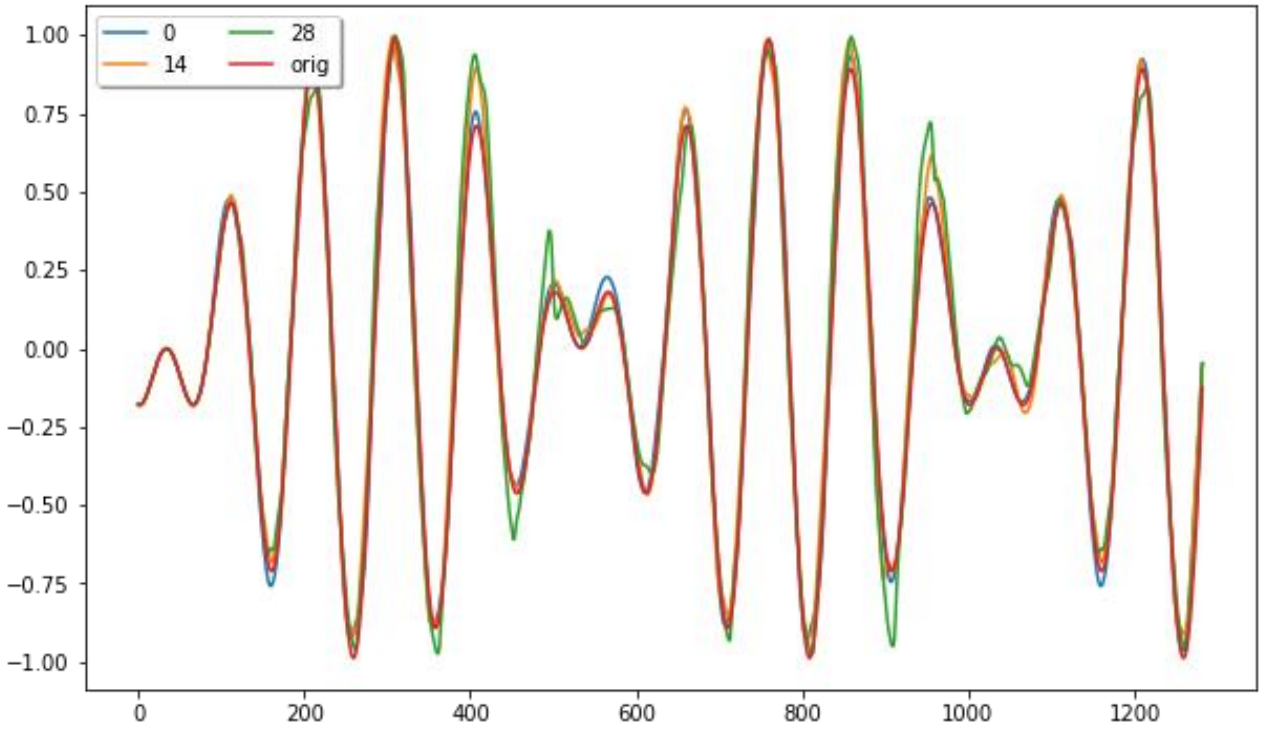

Figure 59: Prediction results

10.2.4.2 Designing LSTM for multivariate data and multiple time steps predictions

In previous sections, each sample of the input data series have been composed by a single feature. Multivariate data series consist of the results of the measurement of many different variables of the same phenomena from different sources and, normally, with different nature. In fact, each sample doesn't have just two dimensions but also a depth.

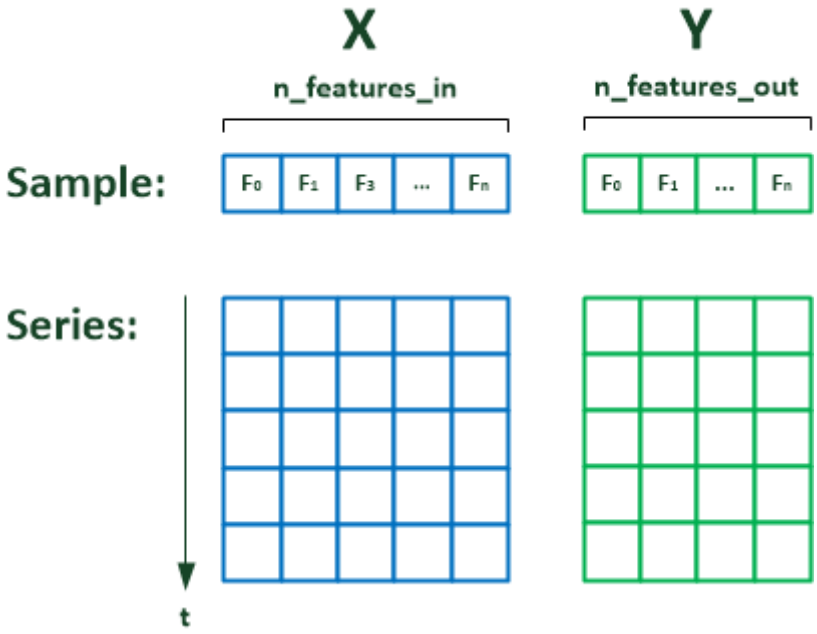

Figure 60: Multivariate time series

Two vectors are required as the main input and are demanded as consistently available. The first one is named X and represents the values sampled from different sensors in a timely consistent manner. The second is then its corresponding target vector Y, representing the faults events.

In the data formatting phase a reshaping action on input data is required, in order to into the transform them in the proper format, as described in section 10.2.4.1.1 for univariate series. This action becomes more challenging and the dataset matrix is then be extended through its full dimension, comprehensive of the input features numbering (named n_features_in).

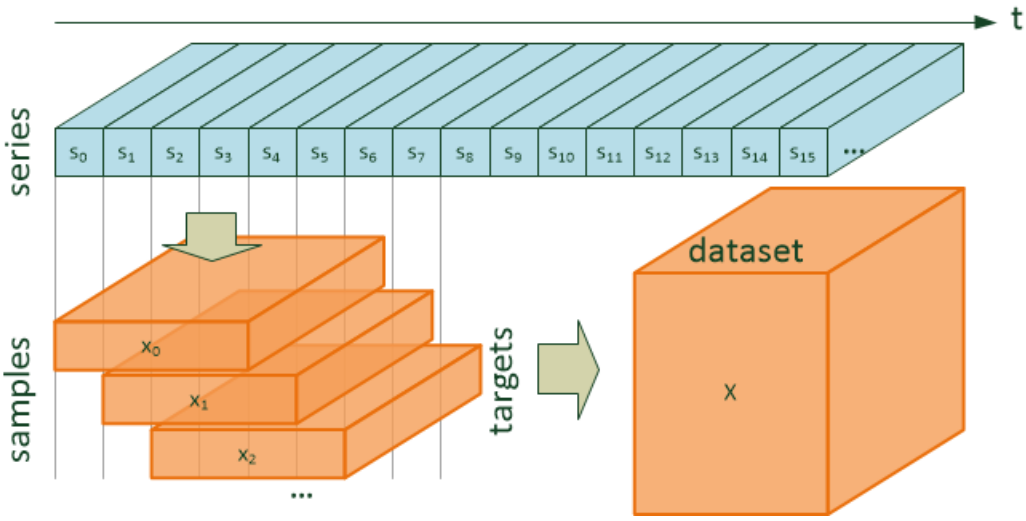

Figure 61: Construction of a dataset from a time series (n_ts_is = 4)

In Figure 62 it is shown the resulting matrixes from the data formatting process on both X and Y vectors.

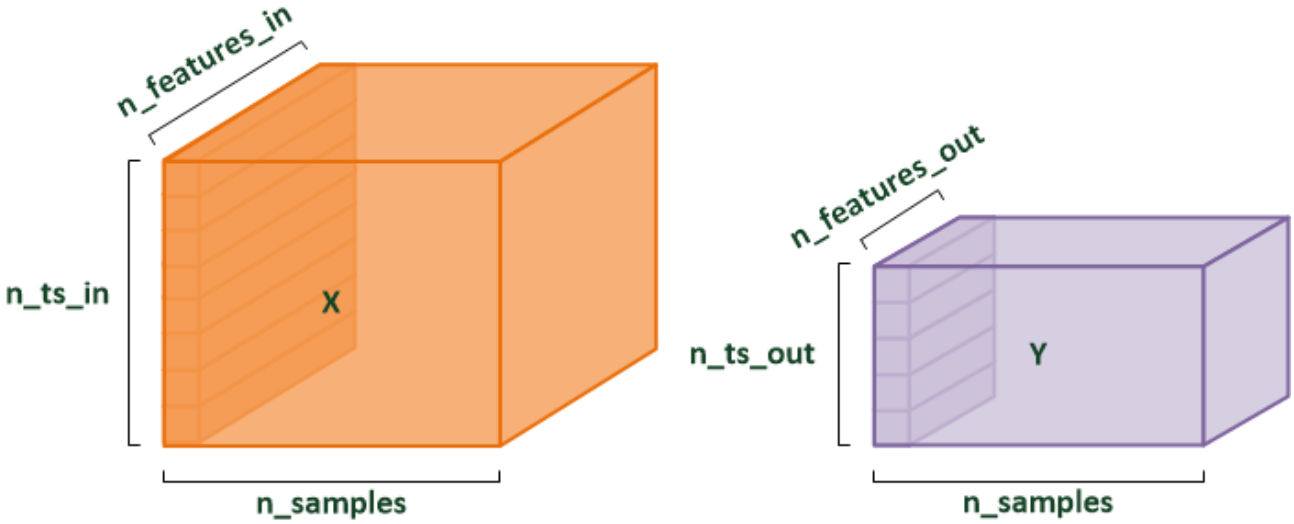

Figure 62: Multivariate dataset

Each input feature can be sampled from different independent sensors and can have different ranges. The contribution of each feature to network's training can vary, depending on its variability that is relative to other features. In order to avoid mismatches in the matrix weights, when it comes to discordancy features, in relation to the same mode, it is fundamental to check if one variance of a single feature has a value that differs of many orders of magnitude from the others. In order to prevent this situation, it is useful to normalize the inputs standardizing each value to the same range or the same standard deviation.

10.2.4.2.1 Supervised learning tests on trained dataset from UC-BSL-2

As described in section 6.1, BSL provides a list of structured records, one per row sampled every 5 minutes. Each row contains, in addition to the timestamp, the logs all the blowers of the machine. For each of the blowers, three values are logged:

- The temperature set by the user [°C] (only for Brady oven)
- The measured temperature [°C]
- The output power at the solid state relay of the reflow

Concerning supervised learning tests, Brady oven data have been chosen for the highest numbers of failures (15 failures in the period between 2008 and 2013) provided. The dataset has been created using only the first 20 blowers' values because the other values were negative or zeros and therefore not pertinent to perform a correct evaluation of machinery's status.

The following parameter values are common all the predictive maintenance experiments. The raw values are assembled into samples as:

- n_features_in = 62
- n_features_out = 1
- n_ts_in = 512 timesteps
- n_ts_out = 1 timestep

As depicted in Figure 63, the first 60 features are the measurements logged by the 20 blowers, relative to the Brady oven. The 61th feature is an index value that represents the temporal distance of the samples from the last fault. In fact, each new sample received without fault increases this value by one. The last feature is a binary value that identifies weather or not a fault has occurred in current sample.

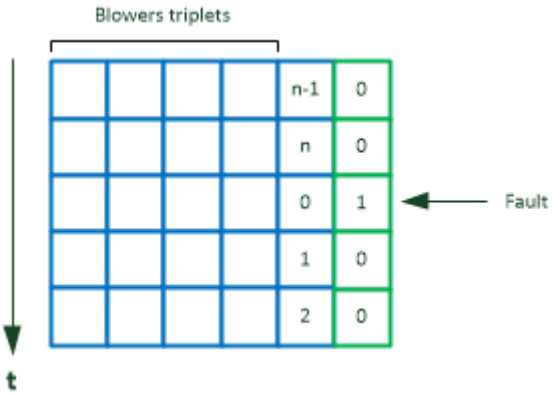

Figure 63: Training features

The used neural network model with three hidden LSTM layers of size 64, 32 and 16 respectively, separated by a repeat vector layer (as described in section 10.2.4.1.3), with an hyperbolic tangent as activation function. The output layer used in this experiment is a fully connected single neuron linear layer.

The network predicts a single scalar output in range [0,1] which represents the probability of a fault in the next 256 time steps (about 21 hours). Figure 64 shows the resulting matrixes structures used for all the experiments.

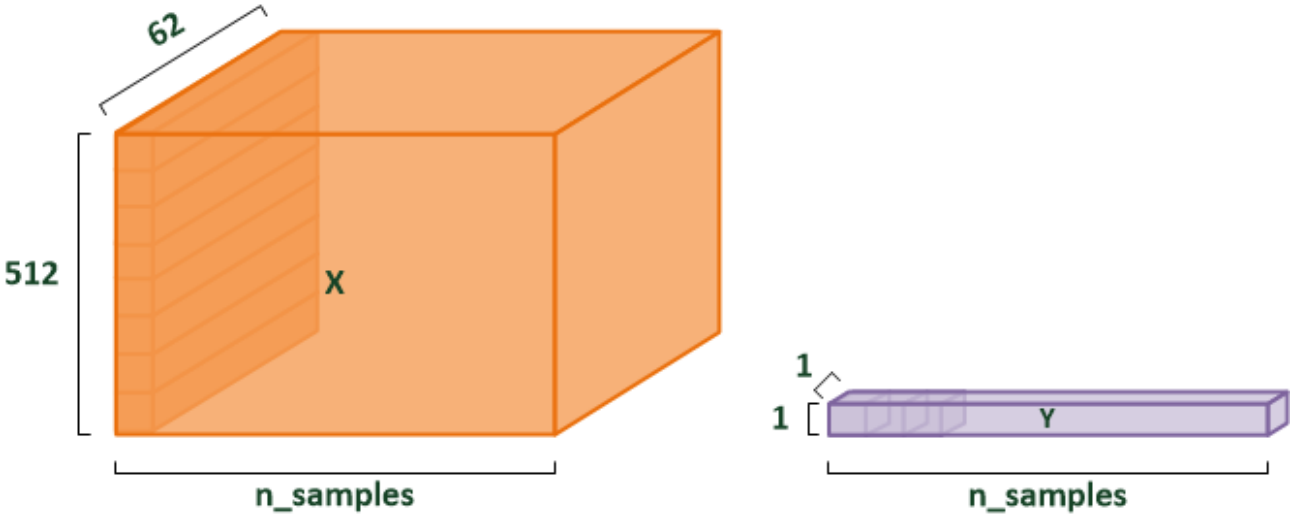

Figure 64: Input matrices for supervised learning test

10.2.4.2.1.1 Test round 0

The first test has been performed leveraging on the full extension of the input dataset. The training phase has been performed on 617110 samples. The n_ts_in values has been decreased to 32 to cope with the big amount of data.

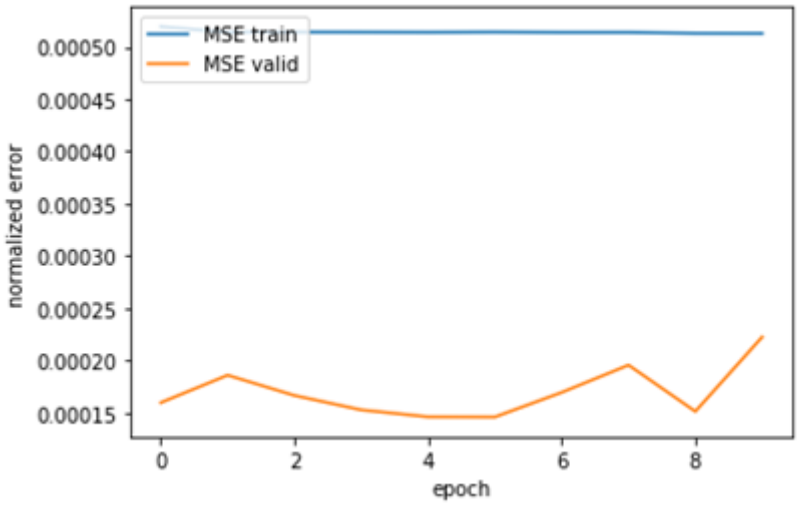

Figure 65: complete dataset training results

As depicted in Figure 65 the network converges immediately with a very low Mean Square Error (MSE) in both validation and training phases. This happen because the number of failures events is way too low: in the best case 15 failures over more than 500K samples are provided. As a consequence, the sampling frequency results to be too high, and the single samples becomes irrelevant. The ANN model is therefore inclined to learn that a constant prediction of the value 0 (no fault) is usually correct.

10.2.4.2.1.2 Test round 1

A second experiment has been performed, in order to balance out input classes for providing a more consistent input to the training phase. In fact, as it has been highlighted in the data assessment chapter (6), classes should be balanced as well known in literature (Xu-Ying Liu, 2009). Informed under sampling is the key answer to provide the required balancing in the two classes. In this test, sensors data have been clustered in order to match the cardinality of the faults.

Moreover, as shown in Figure 66. Unclear events are removed at this early stage, in order to eliminate ineligible events. Two different categories has been chosen: in specific: reported faults confused with maintenance activities and correlated sequential faults. Only faults that have happened while the oven was correctly and

normally working at full performance have been kept for data analysis. Hence, concerning the Brady oven, a total of 9 faults events have been recorded as correct and added to the dataset. Sensor data classes has followed the same process: for balancing out the fault class, data coming from the sensors have been recorded in 9 temporal spans, corresponding to 9 sparse intervals, in which no faults have been reported over the whole reporting period.

Figure 66: Event aggregation and selection

Dataset have been then obtained with the very same process described in the previous experiment in section 10.2.4.2.1.1, for each of the selected index. The network trained with the aggregated data foresees a congregated total of 3686 samples for training and 922 samples for validation and is results are depicted in Figure 67.

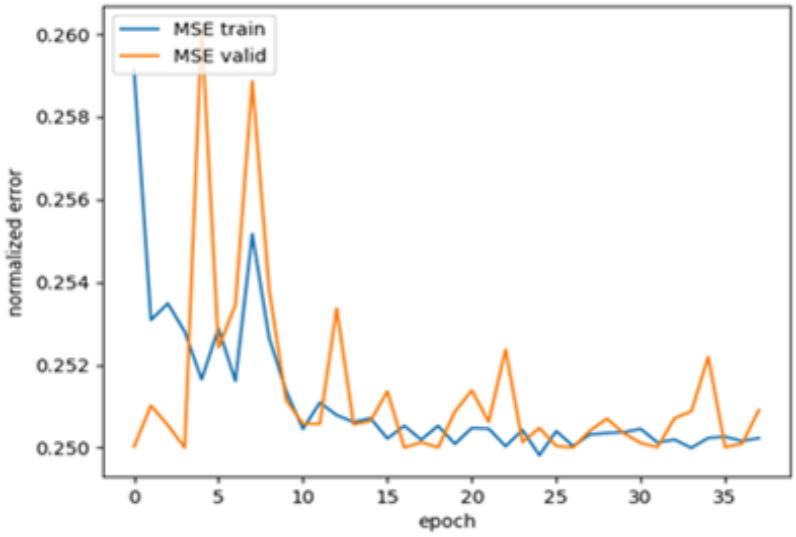

Figure 67: training results for aggregated dataset

As a clear result from the plot, the network fails to converge within the examined time frame. In particular, the achieved normalized error rate did not lead to the expected outcome and is not sufficient to meet the predictive maintenance requirements. While analysing the results, two possible reasons have been found to be responsible for this trend:

- the process of informed under sampling have massively affected the dataset and therefore the brutal change in its cardinality have diminished its meaningfulness;
- for this test, data have not been normalized in order to do not compromise the variance of the input features.

10.2.4.2.1.3 Test round 2

In light of the results of the previous test, the hypothesis of having performed a too deep informed under sampling has been put aside and the second cause has been analysed exclusively in this third experiment. Hence, in order to increase the performance of the Artificial Neural Network, in this third test, all features have been normalized by removing any reference to statistic mean values and scaling to variance values to unit forms. Each column has been therefore affected by these changes, resulting in a completely different dataset in which columns have been centred and scaled independently and each sample have been statistically computed.

Mean and standard deviation statistical values are computed from the training data and used also for the validation dataset normalization. The number of samples is the same of the previous test.

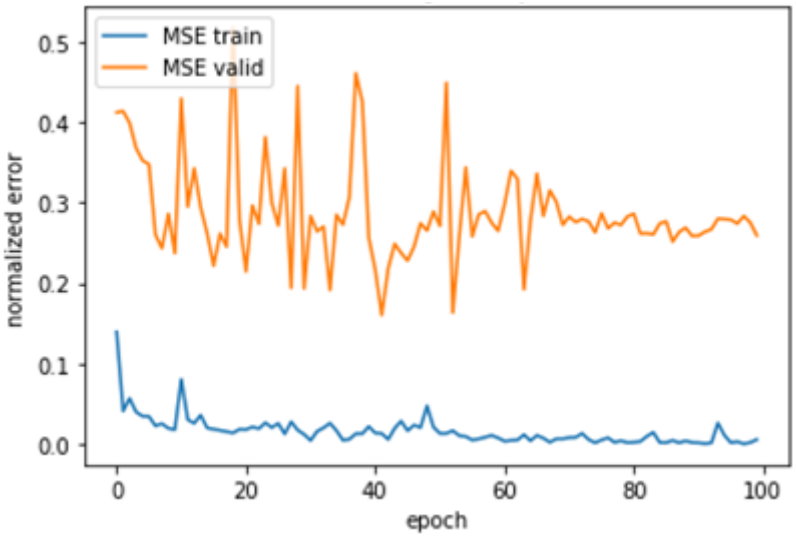

Figure 68: training results for aggregated and normalized dataset

In Figure 68 the blue line is corresponding to the values plotted by the training phase, and its visible how a clear convergence is rapidly achieved after very few epochs. On a further analysis, it is clear how the network learns all available features by memorizing the training dataset in its entirety. So, the model has a high efficiency when it is required to predict the training dataset, but it shows its limitations when required to produce a prediction on data that has never seen before. In fact, results on the validation datasets are predicted with less accuracy has remarked by orange values, in the plot. The blue line is corresponding to the values plotted by the training phase, and its visible how a clear convergence is rapidly achieved after very few epochs. On a further analysis, it is clear how the network learns all available features by memorizing the training dataset in its entirety. So, the model has a high efficiency when it is required to predict the training dataset, but it shows its limitations when required to produce a prediction on data that has never seen before. In fact, results on the validation datasets are predicted with less accuracy has remarked by orange values, in the plot.

This phenomenon is called over fitting and it's a clear consequence of the data subsampling and, most probably, it can be resolved by increasing the number of samples with continuous learning. Nevertheless, for this to happen, more comprehensive data and tests are obviously required to confirm this conclusion which is by the way a strong assumption since it has been demonstrated that this convergence is very likely to happen in a finite time span.

10.2.4.2.2 LSTM deploy

The COMPOSITION ecosystem envisages a lab scale pilot hub for deploying component in a smaller scale controlled environment while they are still on testing. Docker containers are therefore deployed in order to provide a platform for building, distributing and testing components as images by enabling application isolation in a self-contained environment. At the moment, a first version of Deep Learning Toolkit (DLT) container, for Brady oven, has been dockerized and deployed on the COMPOSITION Docker environment. For reference the image is available and tested at:

https://130.192.86.227:8443/#/images/sha256:134d5a71933d6b9f9c0097f16b0d85594b5a22ccc0c88517da51d730b3db3a38/

The trained model provided with the toolkit was created using the process described above and expected results are described in section 10.2.4.1. Moreover, Continuous Learning capabilities have been enabled even in this first version, in order to allow project partners to test out the component and exploit its possibilities.

DLT depends on the following topics to two different topics of the intra-factory MQTT broker:

- COMPOSITION/IntraFactory/DeepLearningToolkit/PredictiveMaintenanceIn to receive samples
- COMPOSITION/IntraFactory/DeepLearningToolkit/ PredictiveMaintenanceOut to publish predictions

A new batch, that must be conformed to the tuples' format and order of the dataset used in the training phase and emphasized in this document, can be received on the input topic. The DLT is expected to publish the latest prediction the output topic, whenever available. The input dataset must be provided according to the format described in section 10.2.4.1.1 as a list of 512 samples with 62 features as a binary Numpy arrays. For a complete data format description please refer to (Scipy reference, n.d.)). This format if for test purposes only and will be changed in future versions of the DLT, most probably aligning it to the project choice of using OGC JSON format for sensors' data.

10.2.4.3 Forecasting with untrained LSTM and continuous learning

Neural network training can be dealt with in different ways depending on data availability. An initial offline training stage is the default strategy when historical data are available. When input data arrives as a live stream, continuous training can be implemented to periodically retrain the network based on recent batches of incoming data.

The two approaches can coexist, but, in case the historical dataset is missing or inadequate, a neural network can be trained exclusively on the go with the continuous learning strategy.

Lack of consistent historical data seems to affect several COMPOSITION use cases, so that it is worth investigating the peculiarity of purely live trained networks.

Contrary to the offline training, in live-only scenarios the networks start untrained, with randomly initialized weights, so that they are unable to deliver significant predictions until a sufficient degree of training is achieved.

So there will be an initial transitional period (aka convergence time), during which the network is to be trained with the live data but the predictions discarded as their accuracy is negligible.

The length of the transitional period is a relevant factor to estimate; unfortunately, it is not easy to determine its value as it typically depends on several parameters including:

- Sampling frequency of the data stream (e.g. how many samples per day)
- How relevant and correlated are the samples and the targets.
- Distribution of the samples in the domain:
    - Uniform distribution is desirable
    - Full domain coverage is desirable
- Distribution of the targets in the codomain
    - Uniform distribution is desirable (e.g. for classification: similar number of targets for all represented classes)
    - Full codomain coverage is desirable (e.g. for classification: all classes are represented)
- Network hyperparameters including network type, layers types, layers numbers, layers order, layers sizes.
- Training hyperparameters including batch size, epochs per batch, learning rate, learning rate adaptation algorithm and parameters.

Even if all these parameters could be prefixed and taken into account simultaneously, the length of the transitional period would not be deterministic all the same. Indeed, the training process is dominated by the stochasticity of live data and of random network initialization resulting in the transitional period being a stochastic variable extremely difficult to characterize.

Anyway, due to its relevance to the projects experiments have been carried out in order to assess for specific and controlled input data, the convergence time of LSTM network for forecasting of univariate time series. This

scenario is relevant to use cases concerning forecasting of prices and of bin fill level. In the following these experiments are described after some additional preliminary considerations.

10.2.4.3.1 Sampling frequency

When forecasting of time series with LSTM is concerned it is important to consider the values of sometime parameters and their ratios. Let's consider the input signal to forecast and its frequency power spectrum describing the distribution of signal's information content.

Since the source signal s(t) is sampled at a specific frequency fsam and the resulting samples are fed to the LSTM network, the latter cannot access the full information content of the original signal. Indeed, the network can leverage the information content of the reconstructed signal sr(t), which is determined by the relationship between the sampling frequency and the maximum frequency of the original signal.

The Nyquist-Shannon theorem asserts that, in order to be able to perfectly reconstruct a limited bandwidth signal, it is sufficient to sample it at a frequency that at least doubles the higher extrema of the bandwidth. In this case, the whole information is preserved in the reconstructed signal.

However, in practice, the only relevant signals with limited bandwidth are sinusoids and their polynomial combinations.

Contrary, a generic signal will have infinite bandwidth. If we are interested in preserving a specific percentage pi of its information for the LSTM network to work on, we can determine analysing the signal's power spectrum, the frequency fmax so the range (0,fmax] Hz contains exactly that percentage of the total power:

$\frac{\int_{-2\pi f_{max}}^{2\pi f_{max}} |S(\omega)|^2 \ d\omega}{\int_{-\infty}^{+\infty} |S(\omega)|^2 \ d\omega} = \frac{p_i}{100}$, where $S(\omega)$is the Fourier transform of the original signal s(t)

Once fmax is determined, the sampling frequency can be chosen to be fsam = 2 fmax.

Alternatively, is the sampling frequency is fixed, the percentage pi can be determined.

In real case scenarios, the function representing the original signal is generally unknown; all the same the consideration exposed before, can allow to evaluate if the sampling frequency is too low with respect to the expected trends of variation of the signal.

10.2.4.3.2 Input time window

After the sampling, the sequence of scalar values is converted into vector samples suitable to LSTM feeding as previously described. The input part of the sample is made of n_ts_in consecutive signal values. That is all the information the LSTM network can use to predict future values and consists in a time window of extension:

$$T_{in} = \frac{n_{ts-in}}{f_{sam}}$$

Standing all the considerations already expressed about n_ts_in dimensioning, additionally it is worth noting that periodic trends in the signal are more and more difficult for the network to learn as their periods get close to (and then bigger than) Tin. Indeed, for the network to learn a trend it is necessary to see it happen in time, but the maximum time length known to the LSTM is Tin. For the sake of clarity, let's suppose that the input signal exhibits two periodic trends: a seasonal one with periodicity of 3 months and an annual one. If we choose n_ts_is so that Tin = 6 month, from each sample the network can see and easily learn the time correlated time structure related to two full periods of the seasonal trend. Contrary each sample only covers half period of the annual trends so that the network always misses the chance to see it and struggle to recognise it a single recurrent trend.

As a consequence, it is reasonable to dimension n_ts_in so that Tin is longer than the period of the longest recurrent trend we expect to have in the series.

Of course, there are two kind of trends that are not possible to include in the input window. These are:

- Noise affecting the signal for any reason. Neural networks anyway are quite good at noise filtering so that this might not be a serious problem in practice.
- Non periodic, long term trend of the function. When not constant this is a trend that is maximally interesting to take into account. Even an optimally trained LSTM will not be able to deliver an accurate prediction across abrupt variations in the long term trend, unless it has been trained on similar events

in the past. The long term trend is innocuous as long as it can be locally considered quasi-constant: its dynamic should be smooth, gradual and slow with respect to Tin. When this assumption does not hold, the best chance is to pre-process the time series by a possible decreasing trend. Therefore, the LSTM does not have learn from (and to predict) actual signal values, but differences between values at consecutive time steps.

In order to confirm what has been said up to this point, and to build additional knowledge, let's proceed with the experiment description.

10.2.4.3.3 Common experimental setup

The following parameter values are common all the experiments.

Each experiment samples a total of n_samples = 20000 raw values from the input time series. Detrending is not applied. The raw values are assembled into samples with:

- inter_sample_stride = 1 timestep
- n_ts_in = 64 timesteps
- n_ts_out = 16 timesteps

This results in a total of 19921 samples.

An initially untrained network for regression is adopted with two hidden LSTM layers of size 16 and 8 neurons respectively, with tanh activation. The output layer is a fully connected single neuron linear layer.

The network is trained live with batches of 32 samples, so that the number of batches is 622. The learning rate adaptation algorithm is RMSprop which is the suggested choice for recurrent neural networks in Keras official documentation.

The different experiments share the same structure: a loop of 20000 iterations. At each iteration the next raw value is acquired and the corresponding sample is generated. The sample is given to the network for prediction and the outcome is saved. Once every 32 iterations, a batch is assembled with the last 32 samples and the network is first evaluated and then retrained with it.

At the end of this loop the following data have been created:

- The sequence of predictions: 19921x16
- The sequence of targets: 19921x16
- The sequence of evaluation metrics, with 622 metric sets (one per batch)

Predictions and targets can be plotted in the same figure to ensure that the former get to better match the latter as time passes.

The series of evaluation metrics allows to create plots that shows the evolution in time of network prediction errors and so to verify if and when the network converges (transitional period evaluation).

The various experiments differ in terms of the generating function of input data, in terms of the initial value of the learning rate, and in terms of the number of training epochs applied to every batch. Concerning these last two hyperparameters, for both of them the expectation is that higher values lead to more exploitation of batch information, so that once reasonable values have been determined ensuring the convergence in the minimal time possible, similar results should be achieved by lowering one of the two values while raising the other.

10.2.4.3.4 Test round 0

In this round of tests, the raw input values are samples from a sinusoid ranging in [-1,1] and with a period T that equals T_{in}. In other words, the input time window T_{in} spans exactly one full period of the trends.

The sinusoid trend has been chosen because it is paradigmatic: it represents the most classical periodic trends, and furthermore, having one single frequency it is easy to ensure the whole information survive the sampling: for example, in this round of tests, $f_{sam} = 16\ f_{max} = 16 * (1/T)$, so that the Nyquist requirement is fully satisfied.

Eight tests have been executed with different hyperparameters values; for each a plot is reported displaying root mean squared error versus time. Considering that the network predicts 16 time steps in the future, it reasonable to expect that the prediction error gets is higher for long term predictions and lower for short terms.

Even if 16 different error trends can be generated (one for each of the prediction intervals), for sake of clarity, in the plots only 3 of them are displayed. In particular prediction at time step 0 (the immediate next one; lowest error), 15 (the most distant in time; highest error) and 7 (midway in between).

In the end a single discussion section wraps up all the tests and their outcomes.

Test 0.0

- Initial learning rate: 0.01
- Training epochs per batch: 1

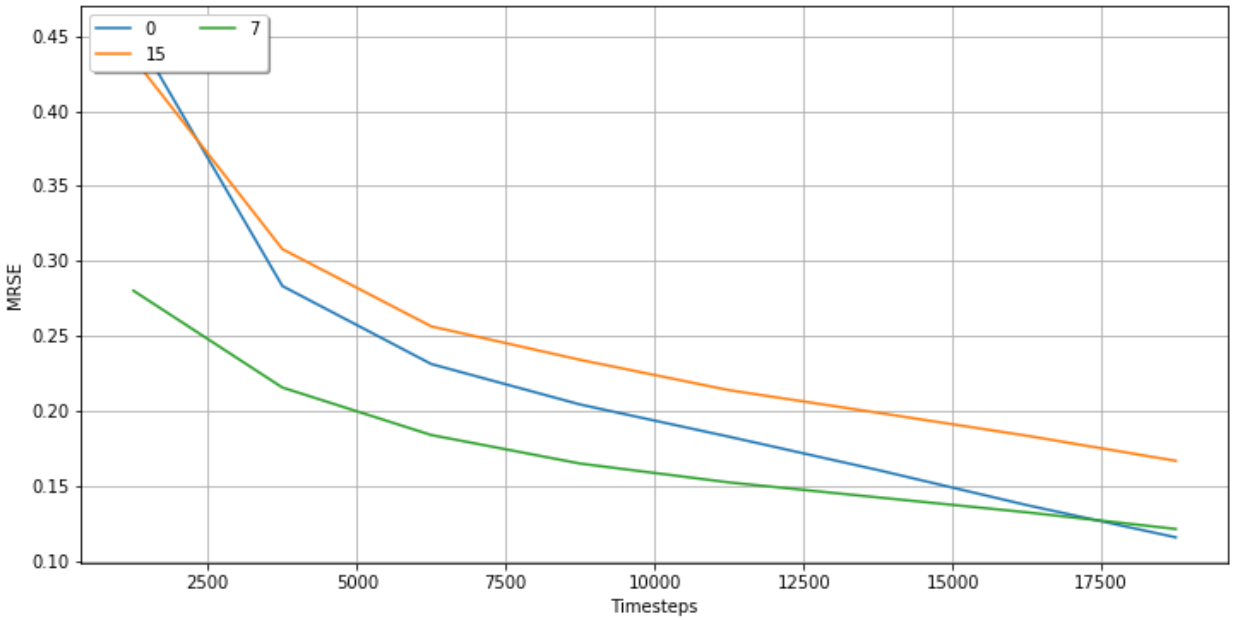

Figure 69: RMSE versus time steps

Test 0.1

- Initial learning rate: 0.01
- Training epochs per batch: 10

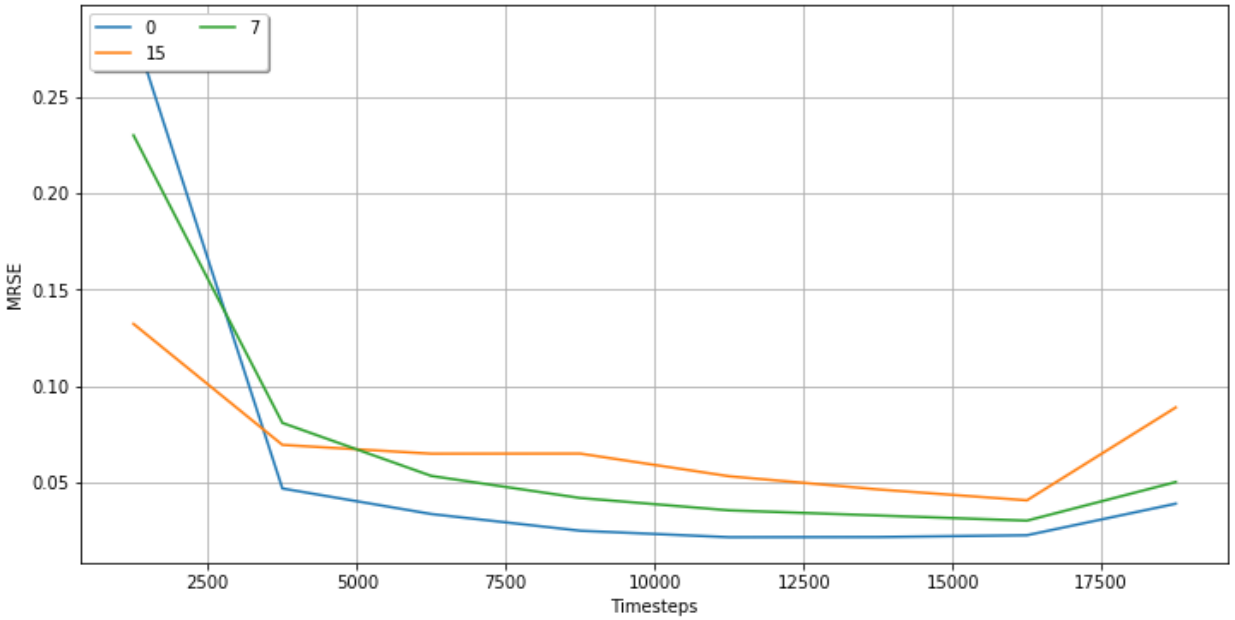

Figure 70: RMSE versus time steps

Test 0.2

- Initial learning rate: 0.01
- Training epochs per batch: 100

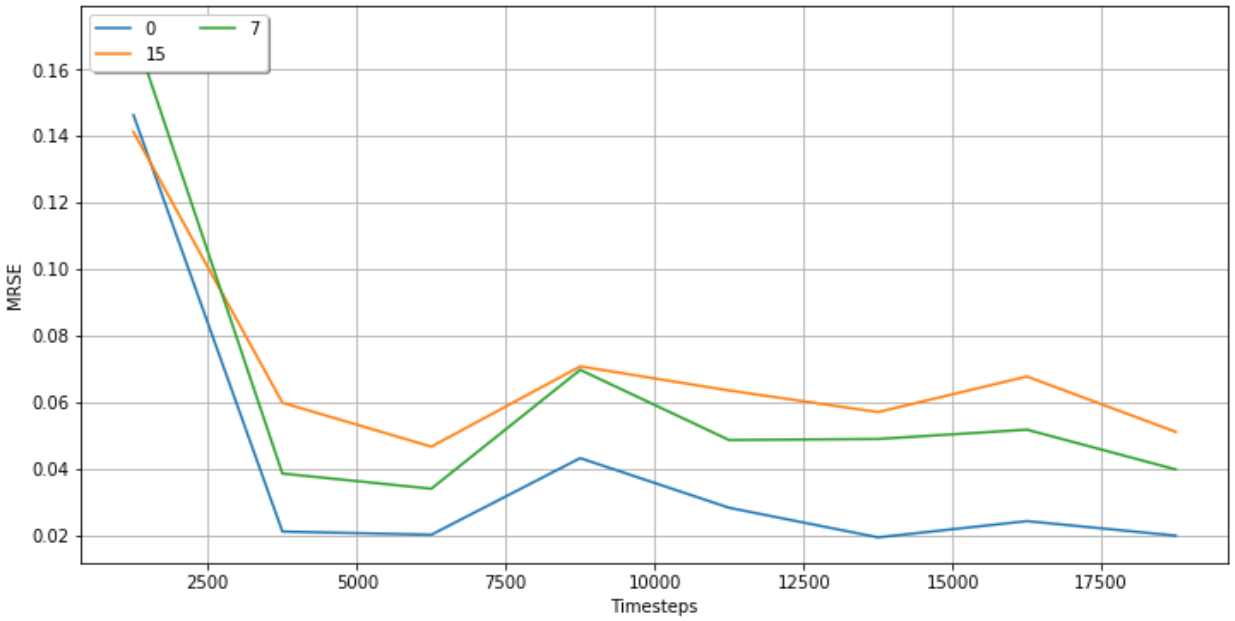

Figure 71: RMSE versus time steps

Test 0.3

- Initial learning rate: 0.1
- Training epochs per batch: 1

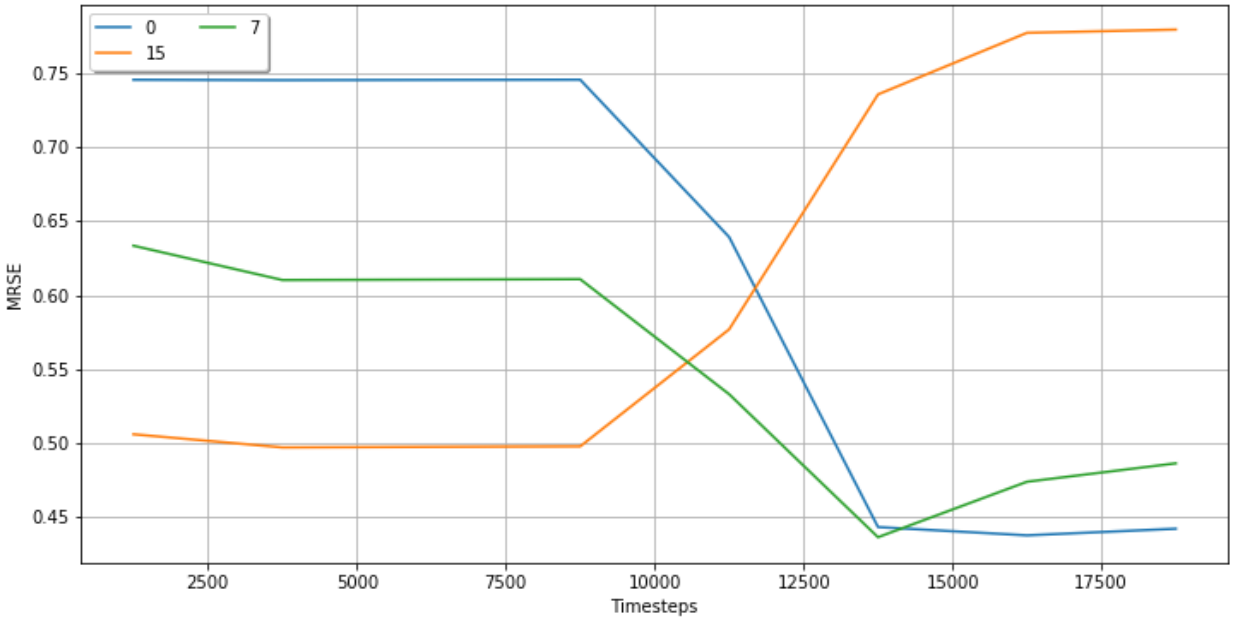

Figure 72: RMSE versus time steps

Test 0.4

- Initial learning rate: 0.1
- Training epochs per batch: 10

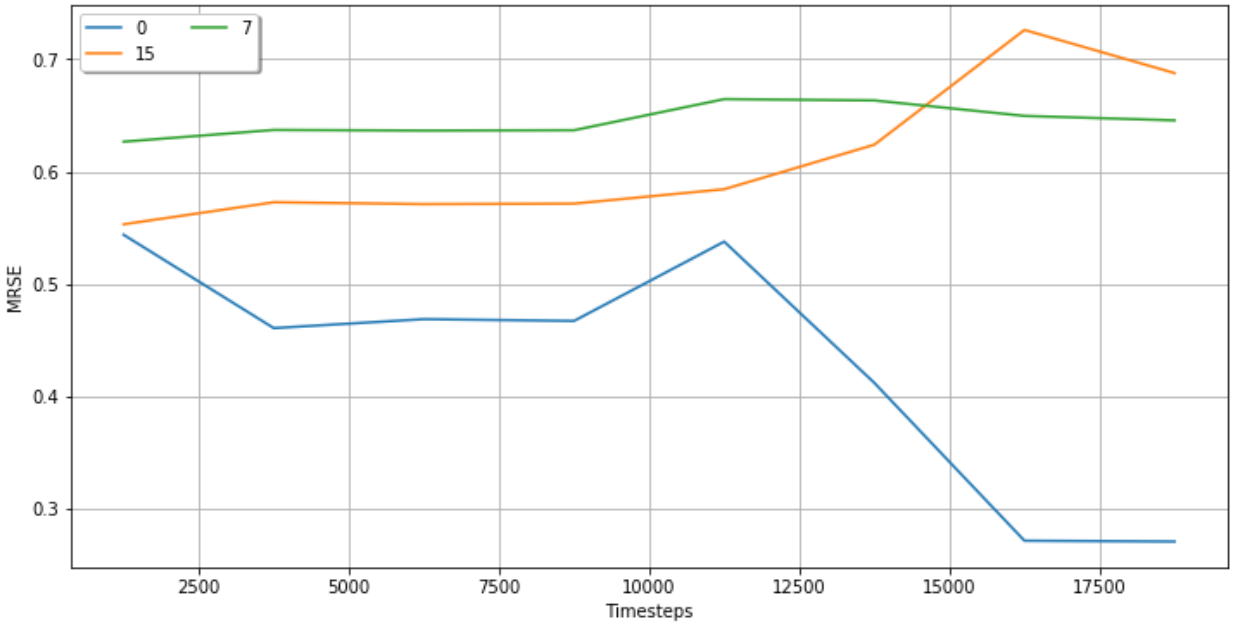

Figure 73: RMSE versus time steps

Test 0.5

- Initial learning rate: 0.001
- Training epochs per batch: 1

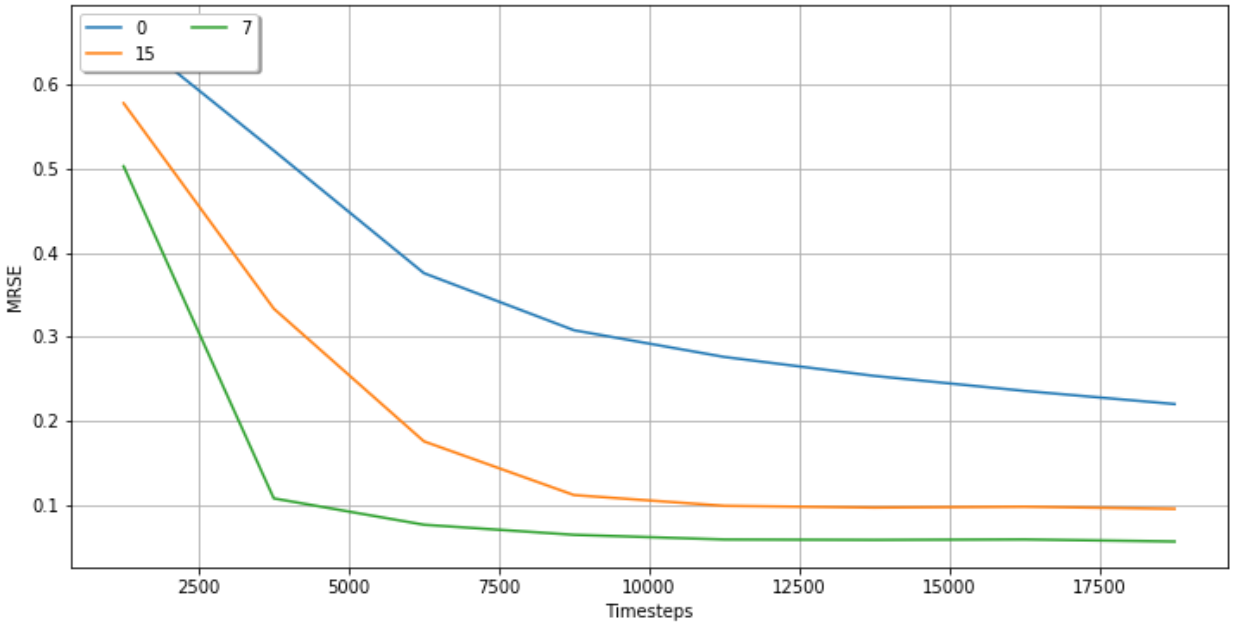

Figure 74: RMSE versus time steps

Test 0.6

- Initial learning rate: 0.001
- Training epochs per batch: 10

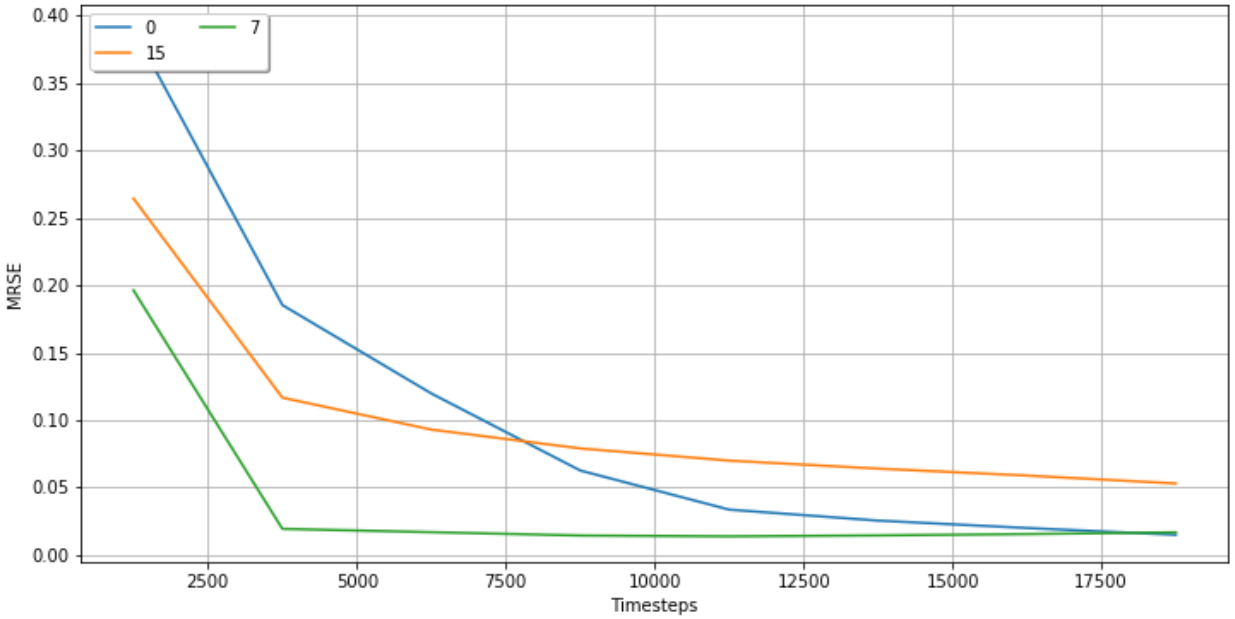

Figure 75: RMSE versus time steps

Test 0.7

- Initial learning rate: 0.03
- Training epochs per batch: 10

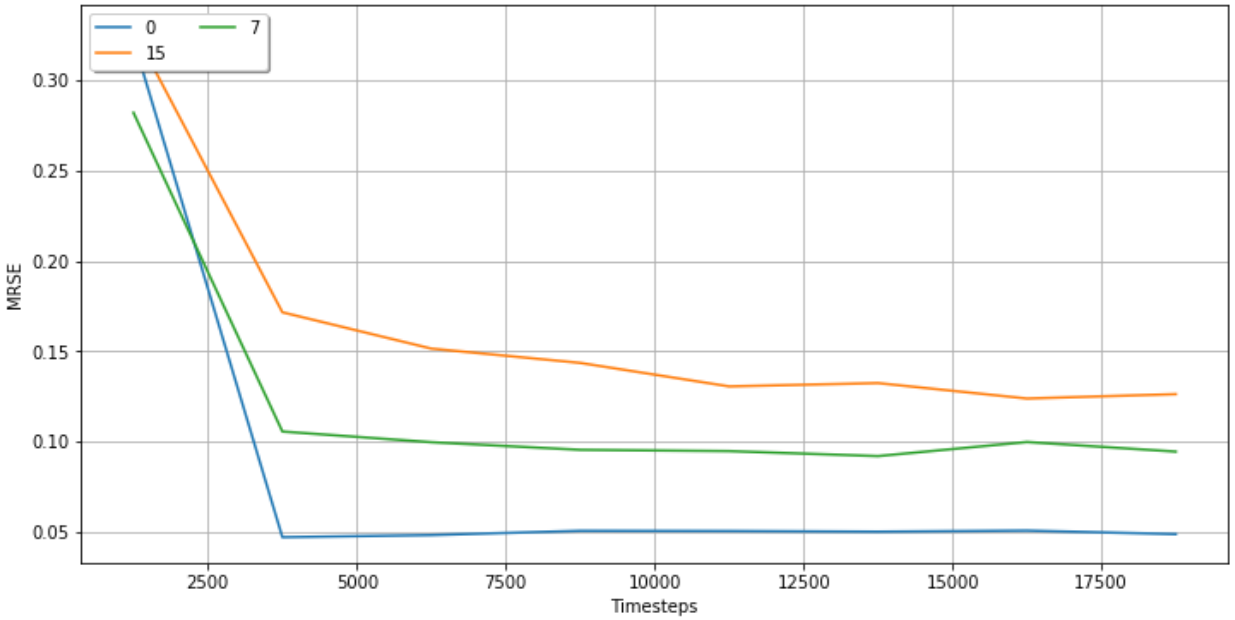

Figure 76: RMSE versus time steps

10.2.4.3.4.1 Results

Table 18 recaps the tests so far detailing the approximate time step of convergence and the related value of RMSE (both evaluated on the next time step prediction, the zero labelled one).

Table 18: test round 0, summary

test id	learning rate	train epochs per batch	ts @ convergence	RMSE @ convergence
0.0	0.01	1	>20000	-
0.1	0.01	10	10000	0.03 (1.5%)
0.2	0.01	100	3700	0.03 (1.5%)
0.3	0.1	1	13700	0.45 (22.5%)
0.4	0.1	10	3700	0.47 (23.5%)
0.5	0.001	1	11700	0.25 (12.5%)
0.6	0.001	10	11700	0.025 (1.25%)

Tests 0.0 to 0.2 adopted an intermediate common value for the initial learning rate: 0.01, and compares different choices of training epochs per batch.

A value of one do not allow the network to fully exploit the information content of the input data so that it fails to converge within the examined period. Values of 10 and 100 lead to convergence: the latter brings earlier convergence but have two drawbacks:

- Potentially it completely overwrites the network weights at each batch (after 100 training steps on the same batch with learning rate of 1/100, all training history previous to the current batch impact about 1% of the weights, while 99% is depends on the current batch). After that, in case new incoming data are similar to older records but do not resembles the ones in the last training batch, the network may behave poorly.
- It is much slower to train.

In both cases the achieved error rate is about 1.5% which can be considered a good result.

These first tests shows that a compromise has to be made in choosing the network substitution rate, which we define as the product of initial learning rate and training epochs. This is just a heuristic consideration because the real learning rate is the result of adaptation algorithm and tends to get lower as the training proceed. Anyway, for the limit case of no adaptation and is classical gradient descent approach, a substitution rate equals or larger than one would imply that training history previous to the current batch is non-significant. This may prevent convergence to ever happen if the input data varies with periodicity greater than the training batch size. This effect can be mitigated, but only to some extent, by increasing the batch size or by including previous batches in training.

Substitution rates lower than one allows for previous training history to survive in the network weights, the lower the rate, the longer the persistence. On the other side, the lower the substitution rate, the longer the convergence interval, and this can prevent network predictions from being effective for very long time.

Tests 0.3 and 0.4 show the effects of high substitution rates: the convergence is achieved is short time, but since network's weights continually lose tracks of the past, the network is unable to cope with new data even if generated by a periodic function. This results in error instability and in general in high error rates.

Tests 0.5 to 0.7 further investigate low and medium substitution rates, which result low errors and long convergence time.

Comparison with test 0.0, whose substitution rate is greater or equal than in test 0.5 and 0.6 shows the stochastic dimension of network training: unexpectedly the latter tests achieve convergence before test 0.0. The conclusion is that it is not possible to determine the best values of hyperparameters for a specific task and network, but only a reasonable range of hyperparameters values which are likely to lead to convergence in a convenient time. The exact amount of convergence time, cannot be guaranteed.

In conclusion, for the specific experimental setup of this tests round, it seems reasonable to choose a learning rate between 0.001 and 0.1, and a number of training epochs per batch of 10. Unfortunately, different setups (input series, n_ts_in, n_ts_out, network size) ale likely to require a re-evaluation of these parameters, which may be onerous to carry out in real case scenarios.

10.2.4.3.5 Test round 1

This second round of test is very similar to the previous one. The only difference is in the period of the input sinusoid, which is modified so that in each sample the network can only see one tenth of the full sinusoidal trend (T_{in} = 0.1T). This is intended to make more complex the training task. Just like in previous round, the sampling does not reduce the amount of information available to the network; the Nyquist theorem is more than respected: f_{sam} = 160 f_{max} .

Only a subset of the most relevant hyperparameters combinations of round 0 has been replicated in round 1. Even if not sequential anymore the minor test numbers have been kept consistent with respect to round 0.

Test 1.0

- Initial learning rate: 0.01
- Training epochs per batch: 1

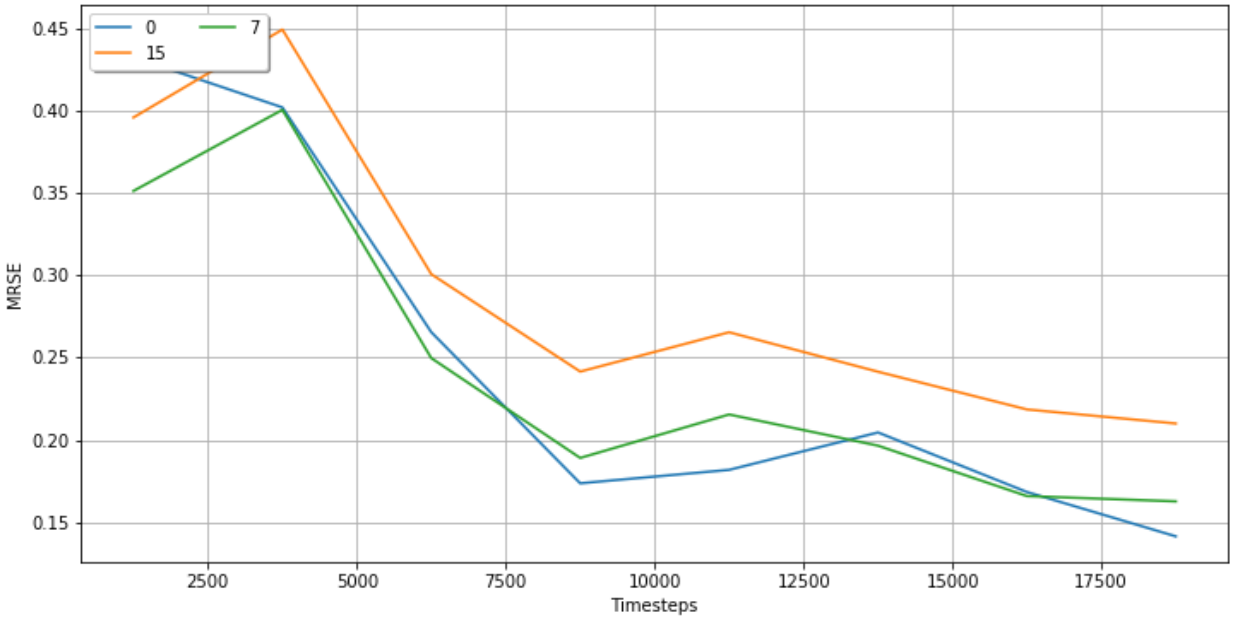

Figure 77: RMSE versus time steps

Test 1.1

- Initial learning rate: 0.01
- Training epochs per batch: 10

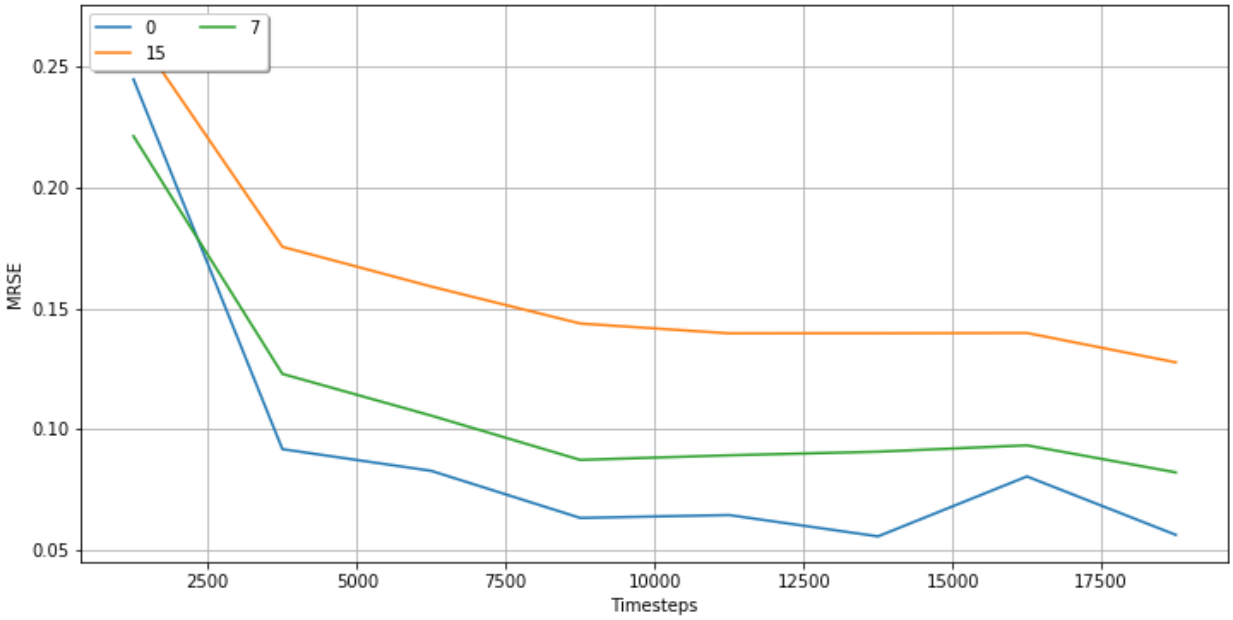

Figure 78: RMSE versus time steps

Test 1.3

- Initial learning rate: 0.1
- Training epochs per batch: 1

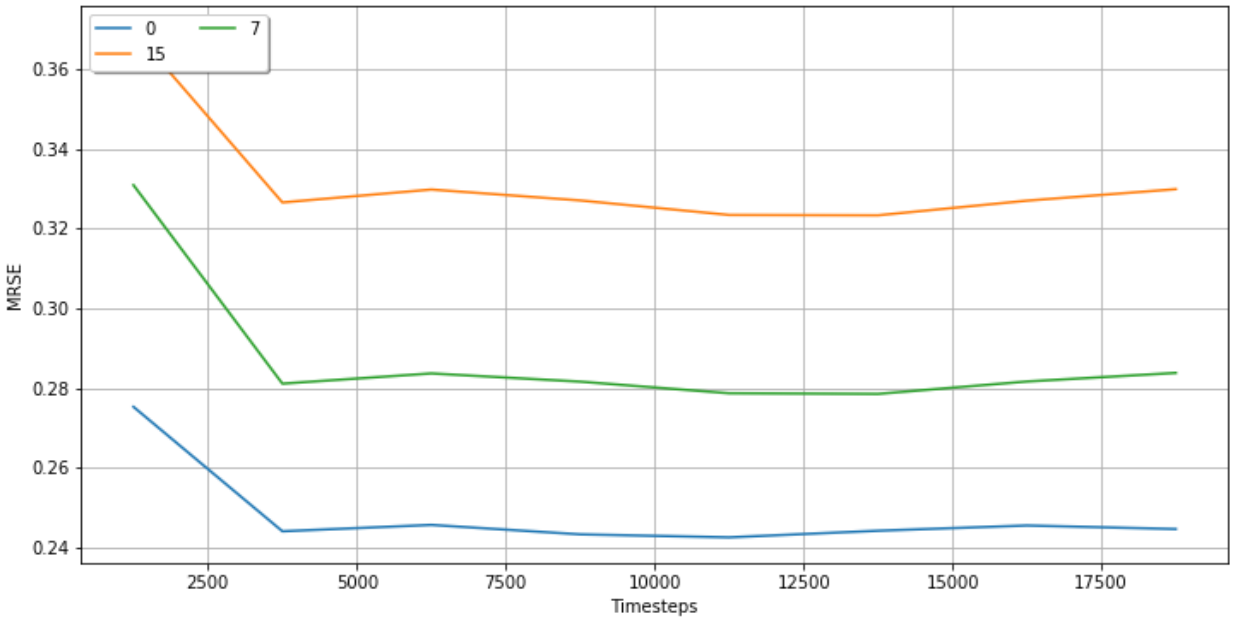

Figure 79: RMSE versus time steps

Test 1.6

- Initial learning rate: 0.001
- Training epochs per batch: 10

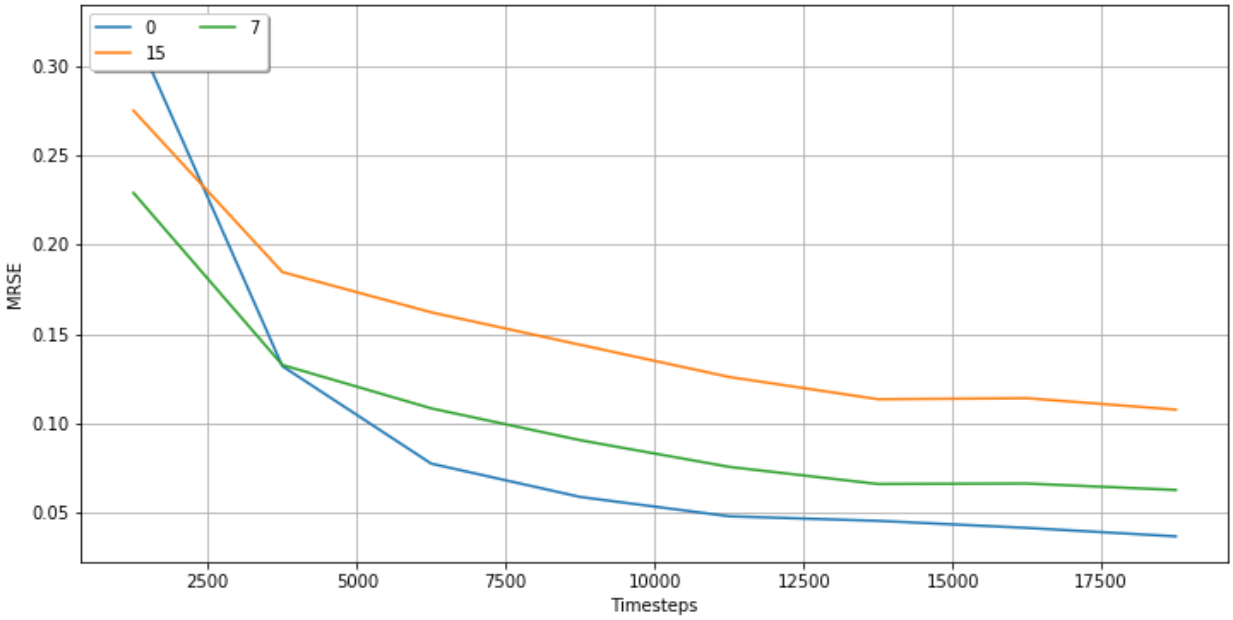

Figure 80: RMSE versus time steps

Test 1.7

- Initial learning rate: 0.03
- Training epochs per batch: 10

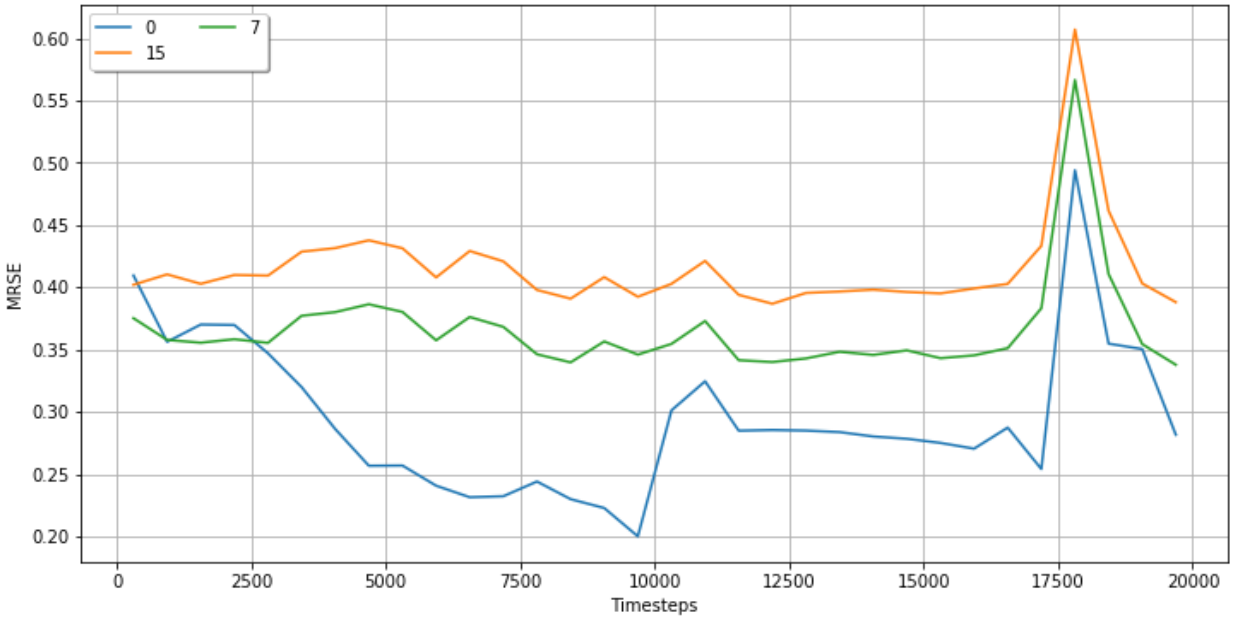

Figure 81: RMSE versus time steps

10.2.4.3.5.1 Results

Table 19 recaps the tests of round 1, detailing the approximate time step of convergence and the related value of RMSE (both evaluated on the next time step prediction, the zero labelled one).

Table 19: test round 10, summary

test id	learning rate	train epochs per batch	ts @ convergence	RMSE @ convergence
1.0	0.01	1	8700	0.18 (9%)
1.1	0.01	10	8700	0.07 (3.5%)
1.3	0.1	1	3700	0.25 (12.5%)
1.6	0.001	10	13700	0.05 (2.5%)
1.7	0.03	10	5000	0.3 (15%)

As expected, by seeing only one tenth at a time of the full input dynamic, the network struggle to consistently reconstruct it. This results in higher error. Hyperparameter choices that were good in tests of round 0, confirm to work at best in round 1. Convergence times seems to be a little shorter, which may be either a random effect or an indirect result of the less average variation per time step of the input signal.

What is more interesting in this round of tests is to compare the plots of target and predictions. The first is of course a perfect sinusoid, while the trend of predictions is less and less accurate as the predictions get further in time.

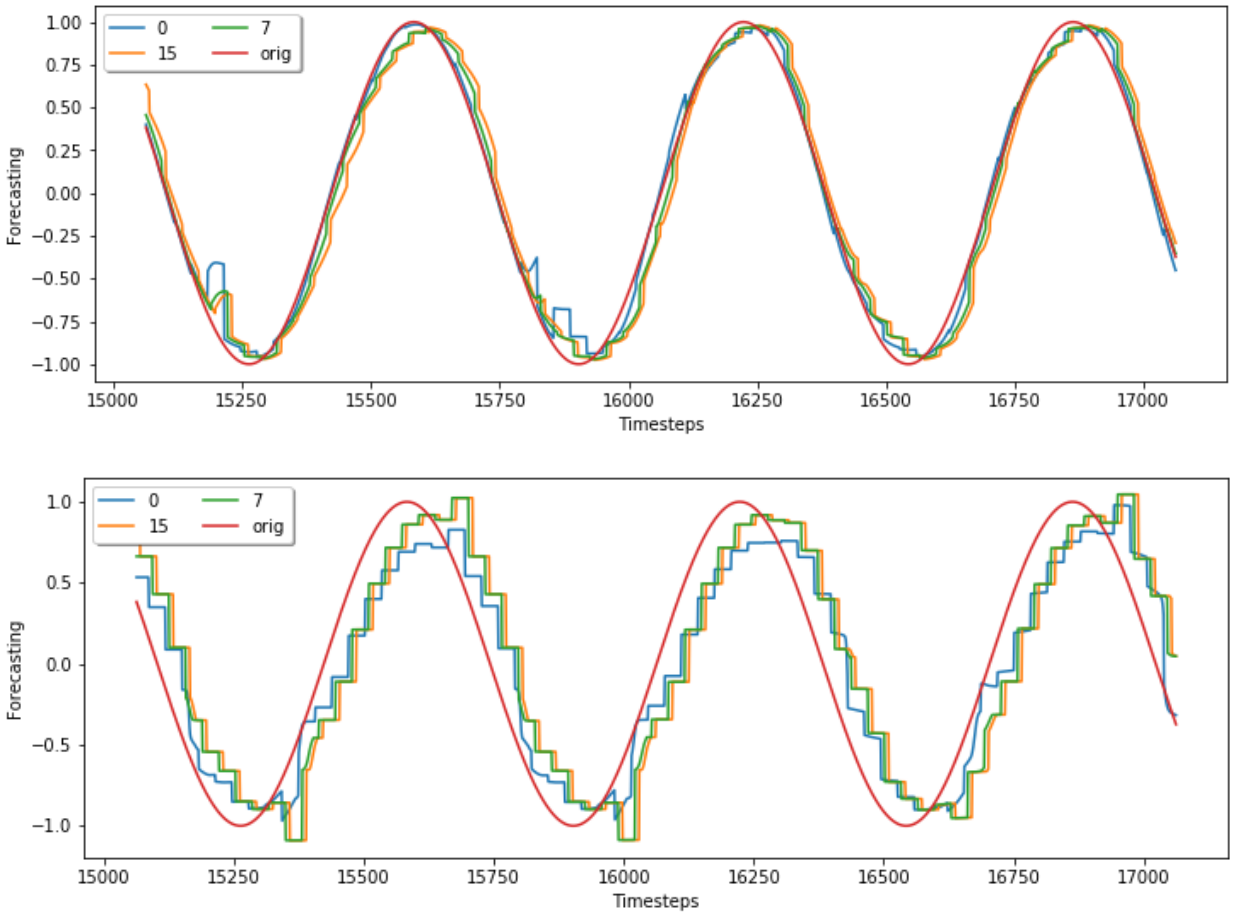

Table 20: target (“orig”) and predictions (“0”, “7”, “15”) at convergence, for tests 1.1 and 1.7.

It’s easy to see that in tests where convergence come with low error like 1.1 and 1.6, the predictions actually match the target quite good, deviations are more visible where the target derivative changes. Contrary, high error tests such as 1.3 and 1.7, due to unfit hyperparameters, shows predictions that are almost constant functions and only the continuous retraining updates the reference value preventing errors to get even higher, and making the curves look like staircases.

10.2.4.3.6 Test round 2

Test round 2 even pushed forward the limitation of the input window: one sample here only covers one 100th of a sin period (T_{in} = 0.01T).

Only one test has been performed having

- Initial learning rate: 0.01
- Training epochs per batch: 10

This is test 2.1 since hyperparameters matches tests 0.1 and 1.1, being one of the best combination of values for the experimental setup.

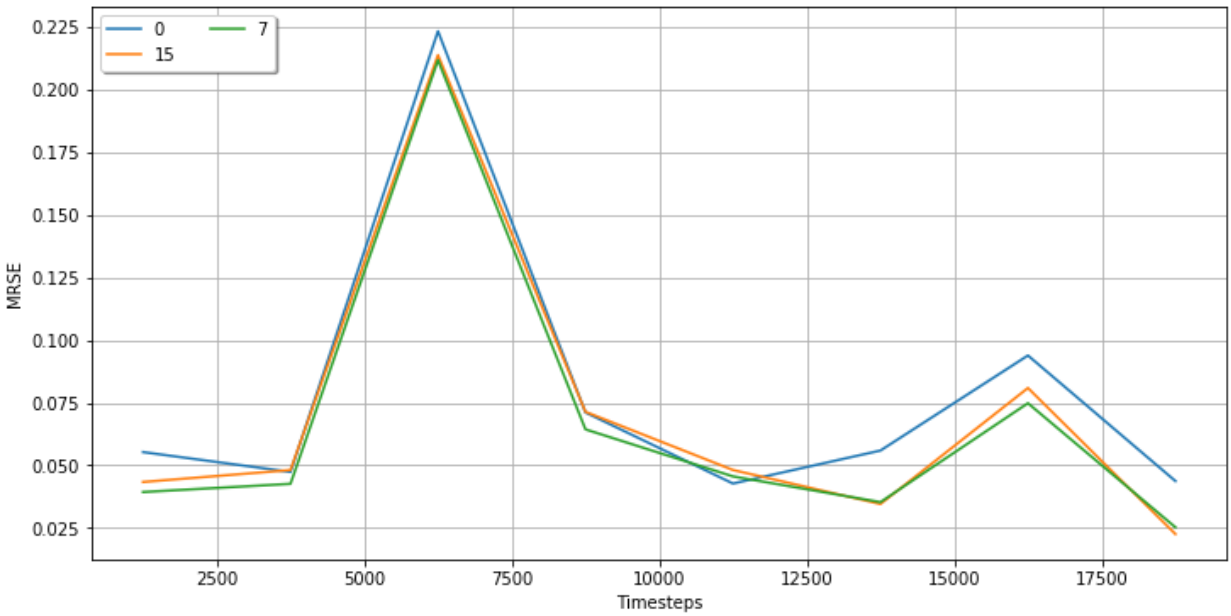

Figure 82: RMSE versus time steps

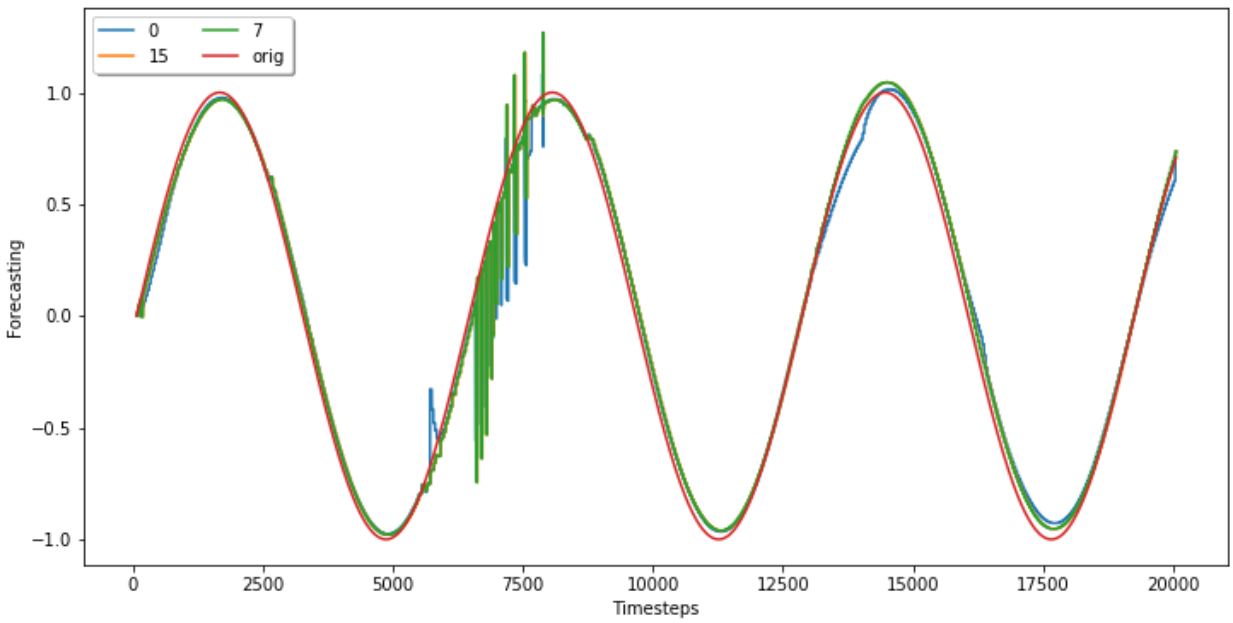

Figure 83: target (“orig”) and predictions (“0”, “7”, “15”) for test 2.1.

Despite the short input time window, RMSE stays quite low at convergence, about 2.5%. Anyway this may be not such the good news it may sound at first. Indeed, the input function now has a very large period, which

means it is almost constant at a local scale (between consecutive time steps, and even along the 16 time steps the network tries to predict). So with this input, high substitution rate is expected to work well as they just focus the network greedily to learn the last batch of samples. Since following samples won't significantly change, this is a good guess.

But this does not mean that the network has converged or is properly trained: the risk is to achieve a network that always predict a constant value based on the most recent training steps, but that whose prediction error raised dramatically in case retraining is not performed and live data start to vary.

This is symptomatic of the impossibility of a neural network to consistently predict about data never seen before (or seen in a remote past and long-time forgotten). This typically result useless network predicting constant values and interpreting all the input dynamic as a noise, which unfortunately means high prediction errors.

When the visibility is so limited with respect to the periodicity of the trend (or if the trend is not periodic at all), detrending strategies might be applied to the raw time series as a pre-processing, before feeding them to neural networks.

10.2.4.3.7 Test round 3

The next step was to test the network with a slightly more complex periodic input. The product of two sinusoids was chosen with different periods, T_{min} and T_{max}. They were dimensioned so that a single sample spanned over 5 short periods and over one tenth of long one: ($T_{in} = 5\ T_{min}$, $T_{in} = 0.1T_{max}$).

Again the single test 3.1 was executed, to validate the choice of hyperparameters that emerged as one of the best from the previous rounds:

- Initial learning rate: 0.01
- Training epochs per batch: 10

As testified by Figure 84 and Figure 86, convergence was achieved around time step 10000 with a RMSE for the next step prediction of 0.025 (1.25%) which is very good.

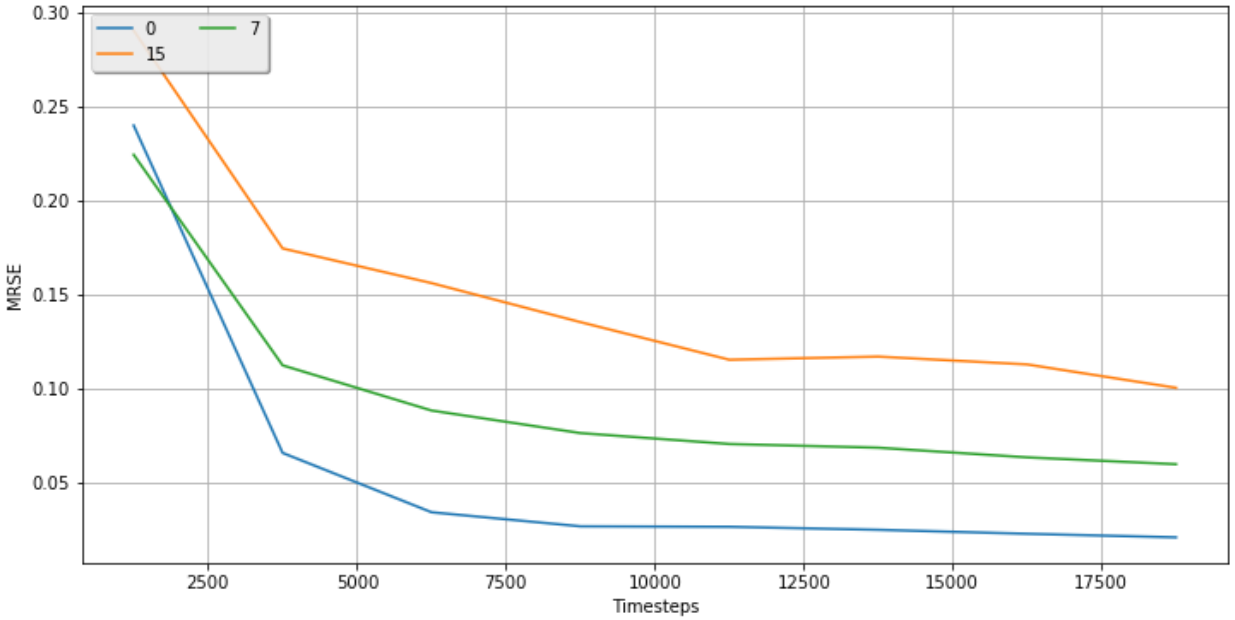

Figure 84: RMSE versus time steps

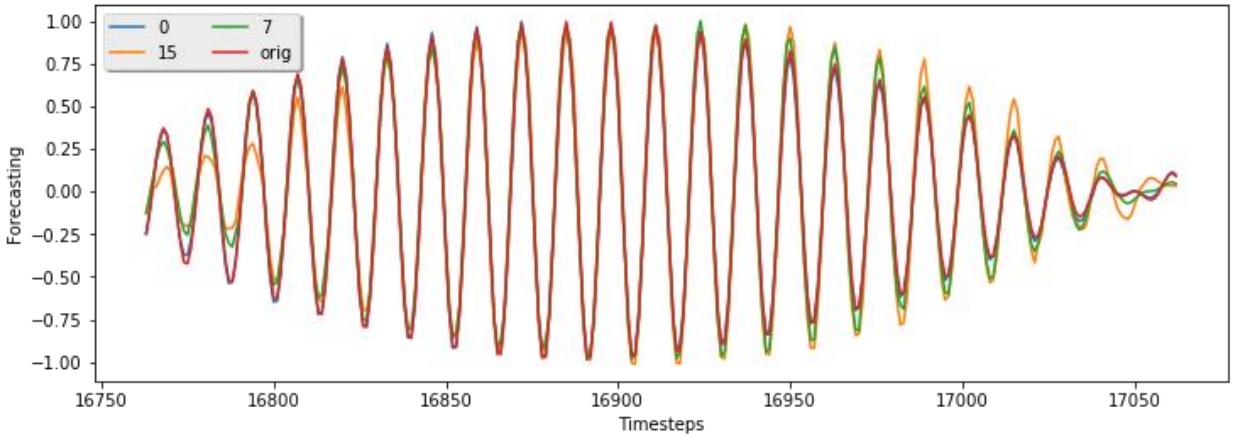

Figure 85: target (“orig”) and predictions (“0”, “7”, “15”) for test 3.1.

The conclusion is that for combination of simple periodic trends, the untrained network with the specified hyperparameters is able to converge in an acceptable time frame achieving low prediction errors, provided that the sampling frequency does not under sample the input signal and that the input time windows does not get too smaller than $0.1T_{max}$.

10.2.4.3.8 Test round 4

This single test round is almost identical to the previous one, the only exception is that the input signal is corrupted by and additive uniform noise in range [-0.15, +0.15], that is the 30% of signal range.

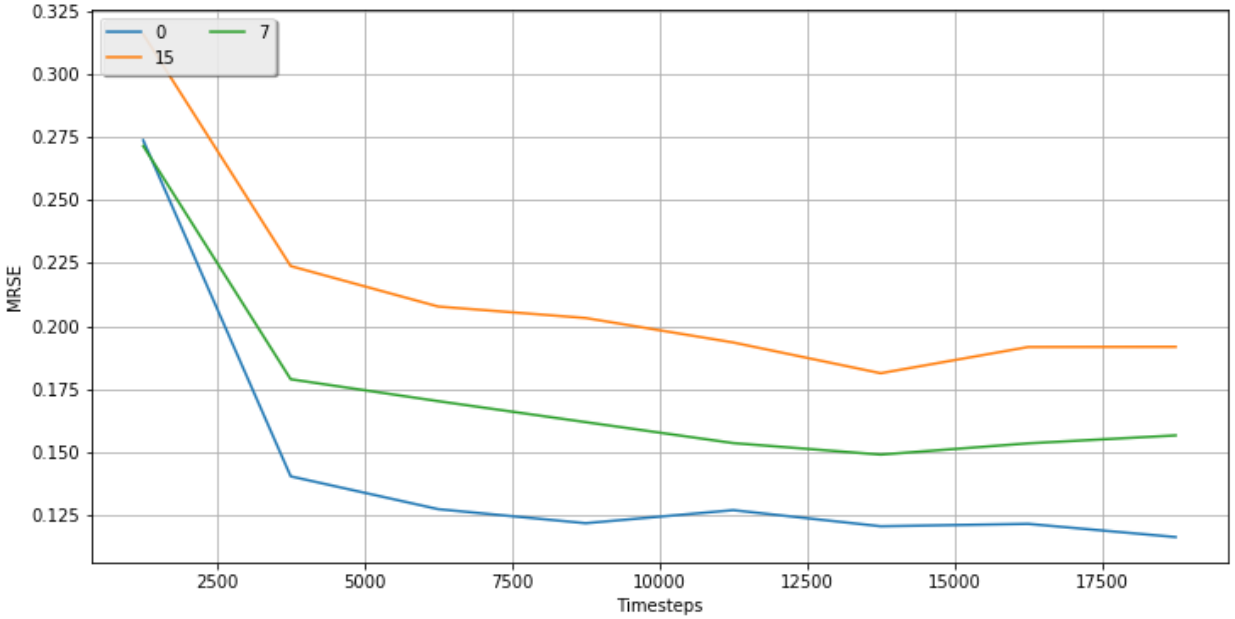

Figure 86: RMSE versus time steps

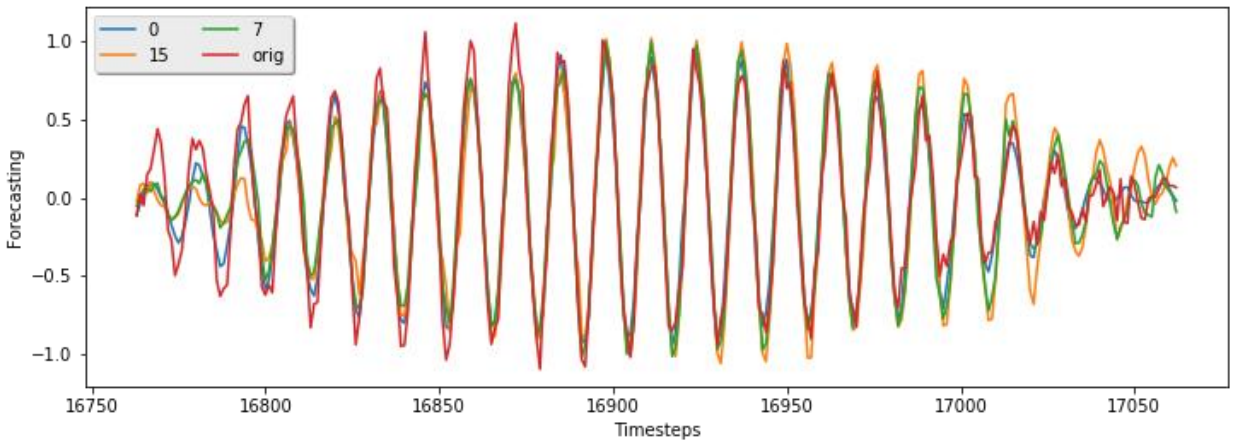

Figure 87: target (“orig”) and predictions (“0”, “7”, “15”) for test 4.1.

The results displayed by Figure 83 and Figure 85 show that convergence is achieved in the same timeframe as is the test without noise, bt with a penalty in terms over RMSE that raises considerably from 0.025 to 0.125 with an increment of 0.1 that corresponds to 5%.

This is value is just slightly larger than the standard deviation of the additive noise alone which is about 0.086, so that the network only adds 0.014 of prediction error with respect to the sum of the previous error and the noise intrinsic component.

In conclusion, this is a good result, showing that the network is able to effectively cope with a considerable amount of noise.

10.2.4.3.9 Final considerations

This preliminary corpus of tests over univariate time series regression with untrained LSTM network suggests that LSTM have the potential to work with time series even starting from an untrained situation, and converge in limited timeframe by means of online learning.

Anyway due to the great variety of hyperparameters involved and to the stochastic nature of the transitional period, it is necessary to expand the experimentation in order to ensure that these encouraging results are confirmed in front of more challenging input series.

In particular, since COMPOSITION use cases that will need untrained forecasting are concerned with metal scrap prices and with bin fill level, it is advisable to perform additional experiments with input series as similar as possible to the expected ones for these use cases.

For the first one, public database of price trends could be used such as those available from London Metal Exchange (London metal dataset, n.d.) also for continuous learning, as already done for univariate series.

For the second a synthetic series could be created with a saw tooth overall shape, controlled by randomized parameters (e.g. for slope and bin full period).

11 Annex II

Rhythmia oven provides one event file per day (e.g. "Dec 01 2013 event.txt"). Each file contains the list of events occurred to the oven. Each event has a timestamp and a textual description. The events should be read line by line.

In the following the events discrimination action performed in T5.2 is reported, as consistent required input from the DLT. It consists in a set of textual events must align to the rules proposed in the table below.

The columns description follows:

- DLT input position: is the required position of the field in the DLT samples table. The first 40 values of the table are reported here.
- Value type: textual description of the field's content.
- Event row example: a textual description of the event as provided in the data file as an example.
- Mapping required action: rule that needs to be followed to comply with DLT required input.

Table 21: Example of logs event

DLT input position	Value type	Event row example (*)	Mapping required action
40	Oven Operate Mode	4 OVEN 10/14/2011 9:03:04 AM Switch to Operate Mode	1 when the event is raised, 0 otherwise
41	OvenCooldownMode	4 OVEN 10/14/2011 9:03:04 AM Switch to Cooldown Mode	1 when the event is raised, 0 otherwise
42	Oven Edit Mode	4 OVEN 10/14/2011 9:03:04 AM Switch to Edit Mode	1 when the event is raised, 0 otherwise
43	Nitrogen status OFF	99 OVEN 3/23/2012 12:46:42 PM Nitrogen was turned ON	1 when the event is raised, 0 otherwise
44	Nitrogen status ON	99 OVEN 3/23/2012 12:46:42 PM Nitrogen was turned OFF	1 when the event is raised, 0 otherwise
45	Nitrogen low state	126 OVEN 3/23/2012 2:30:39 PM Nitrogen at low flow state	1 when the event is raised, 0 otherwise
46	Nitrogen normal state	126 OVEN 3/23/2012 2:30:39 PM Nitrogen at normal flow state	1 when the event is raised, 0 otherwise
47	Nitrogen high state	126 OVEN 3/23/2012 2:30:39 PM Nitrogen at high flow state	1 when the event is raised, 0 otherwise
48	Nitrogen flow is off	222 OVEN 5/24/2011 7:06:47 AM Nitrogen flow is off	1 when the event is raised, 0 otherwise
49	Cooldown will load	36 OVEN 11/09/2014 20:47:08 Cooldown will load, waiting for boards to clear oven	1 when the event is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
50	Emergency stopcooldown	3 OVEN 10/16/2011 11:42:49 AM Emergency Stop button was pressed! Cooldown loaded	1 when the event is raised, 0 otherwise
51	Exhaust insufficient cooldown	6 OVEN 11/30/2011 7:45:14 AM Exhaust is insufficient and the oven will load Cooldown	1 when the event is raised, 0 otherwise
52	High water temperature cooldown	287 OVEN 5/25/2011 3:35:00 AM High Water Temp Alarm Cool Down Loaded!	1 when the event is raised, 0 otherwise
53	< COOLDOWN>	35 OVEN 26/04/2015 22:48:44 < COOLDOWN>	1 when the event is raised, 0 otherwise
54	PPM Level within Alarm1 warning	4 OVEN 9/30/2011 6:51:49 AM PPM Level within Alarm1 warning	1 when the event is raised, 0 otherwise
55	PPM has exceeded the amount set in alarm1	6 OVEN 9/30/2011 7:12:17 AMOxygen PPM has exceeded the amount set in alarm1	1 when the event is raised, 0 otherwise
56	PPM Level within Alarm2 warning	4 OVEN 9/30/2011 6:51:49 AM PPM Level within Alarm2 warning	1 when the event is raised, 0 otherwise
57	PPM has exceeded the amount set in alarm2	5 OVEN 6/20/2011 3:53:17 PM Oxygen PPM has exceeded the amount set in alarm2	1 when the event is raised, 0 otherwise
58	Heat 1 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 1: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
59	Heat 2 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 2: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
60	Heat 3 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 3: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
61	Heat 4 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 4: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
62	Heat 5 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 5: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
63	Heat 6 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 6: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
64	Heat 7 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 7: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
65	Heat 8 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 8: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
66	Heat 9 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 9: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
67	Heat 10 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 10: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
68	Heat 11 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 11: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
69	Heat 12 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 12: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
70	Heat 13 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 13: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
71	Heat 14 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 14: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
72	Heat 15 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 15: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
73	Heat 16 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 16: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
74	Heat 17 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 17: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
75	Heat 18 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 18: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
76	Heat 19 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 19: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
77	Heat 20 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 20: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
78	Heat 21 hi warning	20 OVEN 15/12/2016 09:26:32 Heat 21: Hi Warning 172°C	the warning temperature value (172 in the example) when the warning is raised, 0 otherwise
79	Heat 1 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 1: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
80	Heat 2 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 2: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
81	Heat 3 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 3: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
82	Heat 4 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 4: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
83	Heat 5 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 5: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
84	Heat 6 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 6: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
85	Heat 7 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 7: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
86	Heat 8 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 8: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
87	Heat 9 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 9: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
88	Heat 10 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 10: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
89	Heat 11 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 11: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
90	Heat 12 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 12: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
91	Heat 13 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 13: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
92	Heat 14 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 14: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
93	Heat 15 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 15: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
94	Heat 16 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 16: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
95	Heat 17 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 17: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
96	Heat 18 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 18: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
97	Heat 19 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 19: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
98	Heat 20 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 20: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
99	Heat 21 lo warning	271 OVEN 02/12/2016 17:01:55 Heat 21: Lo Warning 195°C	the warning temperature value (195 in the example) when the warning is raised, 0 otherwise
100	Heat 1: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 1: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
101	Heat 2: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 2: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
102	Heat 3: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 3: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
103	Heat 4: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 4: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
104	Heat 5: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 5: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
105	Heat 6: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 6: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
106	Heat 7: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 7: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
107	Heat 8: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 8: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
108	Heat 9: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 9: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
109	Heat 10: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 10: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
110	Heat 11: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 11: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
111	Heat 12: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 12: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
112	Heat 13: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 13: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
113	Heat 14: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 14: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
114	Heat 15: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 15: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
115	Heat 16: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 16: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
116	Heat 17: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 17: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
117	Heat 18: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 18: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
118	Heat 19: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 19: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
119	Heat 20: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 20: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
120	Heat 21: Rise Rate Alarm on Channel	113 OVEN 16/02/2016 10:41:10 Heat 21: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
121	Heat 1: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 1: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
122	Heat 2: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 2: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
123	Heat 3: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 3: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
124	Heat 4: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 4: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
125	Heat 5: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 5: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
126	Heat 6: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 6: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
127	Heat 7: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 7: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
128	Heat 8: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 8: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
129	Heat 9: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 9: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
130	Heat 10: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 10: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
131	Heat 11: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 11: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
132	Heat 12: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 12: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
133	Heat 13: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 13: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
134	Heat 14: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 14: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
135	Heat 15: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 15: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
136	Heat 16: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 16: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
137	Heat 17: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 17: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
138	Heat 18: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 18: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
139	Heat 19: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 19: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
140	Heat 20: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 20: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
141	Heat 21: Hi Process Alarm	24 OVEN 07/11/2013 18:26:47 Heat 21: Hi Process Alarm 3277°C	the alarm temperature value (3277 in the example) when the warning is raised, 0 otherwise
142	Heat 1: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 1: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
143	Heat 2: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 2: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
144	Heat 3: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 3: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
145	Heat 4: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 4: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
146	Heat 5: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 5: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
147	Heat 6: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 6: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
148	Heat 7: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 7: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
149	Heat 8: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 8: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
150	Heat 9: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 9: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
151	Heat 10: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 10: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
152	Heat 11: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 11: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
153	Heat 12: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 12: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
154	Heat 13: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 13: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
155	Heat 14: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 14: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
156	Heat 15: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 15: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
157	Heat 16: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 16: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
158	Heat 17: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 17: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
159	Heat 18: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 18: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
160	Heat 19: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 19: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
161	Heat 20: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 20: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
162	Heat 21: Hi Deviation Alarm	33 OVEN 22/11/2014 20:16:18 Heat 21: Hi Deviation Alarm 55°C	the alarm temperature value (55 in the example) when the warning is raised, 0 otherwise
163	Heat 1: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 1: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
164	Heat 2: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 2: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
165	Heat 3: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 3: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
166	Heat 4: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 4: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
167	Heat 5: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 5: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
168	Heat 6: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 6: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
169	Heat 7: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 7: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
170	Heat 8: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 8: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
171	Heat 9: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 9: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
172	Heat 10: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 10: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
173	Heat 11: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 11: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
174	Heat 12: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 12: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
175	Heat 13: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 13: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
176	Heat 14: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 14: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
177	Heat 15: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 15: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
178	Heat 16: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 16: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
179	Heat 17: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 17: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
180	Heat 18: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 18: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
181	Heat 19: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 19: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
182	Heat 20: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 20: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
183	Heat 21: Lo Deviation Alarm	40 OVEN 03/01/2017 08:58:09 Heat 21: Lo Deviation Alarm 190°C	the alarm temperature value (190 in the example) when the warning is raised, 0 otherwise
184	Heat 1: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 1: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
185	Heat 2: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 2: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
186	Heat 3: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 3: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
187	Heat 4: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 4: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
188	Heat 5: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 5: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
189	Heat 6: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 6: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
190	Heat 7: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 7: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
191	Heat 8: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 8: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
192	Heat 9: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 9: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
193	Heat 10: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 10: is	1 when the event is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
		drawing output beyond allowed threshold	
194	Heat 11: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 11: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
195	Heat 12: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 12: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
196	Heat 13: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 13: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
197	Heat 14: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 14: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
198	Heat 15: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 15: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
199	Heat 16: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 16: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
200	Heat 17: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 17: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
201	Heat 18: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 18: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
202	Heat 19: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 19: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
203	Heat 20: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 20: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise
204	Heat 21: Heat drawing beyond allowed threshold	157 OVEN 25/11/2015 14:19:56 Heat 21: is drawing output beyond allowed threshold	1 when the event is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
205	Lube #1 cycle	5 OVEN 17/05/2017 07:40:21 Automatic Lube #1 Cycle Start.	1 when the event is raised, 0 otherwise
206	Lube #2 cycle	6 OVEN 17/05/2017 12:40:21 Automatic Lube #2 Cycle Start.	1 when the event is raised, 0 otherwise
207	Over current draw	83 OVEN 16/05/2017 14:43:14 Over Current Draw!5 or More Zones at 100%	1 when the event is raised, 0 otherwise
208	Broad count reset	14 OVEN 06/06/2017 14:49:04 Board count was reset on Lane 1	1 when the event is raised, 0 otherwise
209	Backup timer activated	73 OVEN 26/01/2016 15:36:08 Backup timer activated, oven control software will continue to operate.	1 when the event is raised, 0 otherwise
210	Recipe type	1 operator 10/13/2012 12:00:04 AM C:\Oven\Recipe Files\Insignia200.JOB	The recipe file path (C:\Oven\Recipe Files\Insignia200.JOB in the example), an empty string otherwise
211	Cool 1 Flux Heater: Lo Warning	159 OVEN 18/04/2015 12:16:20 Cool 1 Flux Heater: Lo Warning 13°C	the temperature value (13 in the example) when the warning is raised, 0 otherwise
212	Cool 2 Flux Heater: Lo Warning	2 OVEN 06/04/2015 03:50:44 Cool 2 Flux Heater: Lo Warning 10°C	the temperature value (10 in the example) when the warning is raised, 0 otherwise
213	Cool 1 Flux Heater: Hi Warning	46 OVEN 01/04/2015 08:53:39 Cool 1 Flux Heater: Hi Warning 14°C	the temperature value (14 in the example) when the warning is raised, 0 otherwise
214	Cool 2 Flux Heater: Hi Warning	109 OVEN 15/05/2015 12:39:53 Cool 2 Flux Heater: Hi Warning 10°C	the temperature value (10 in the example) when the warning is raised, 0 otherwise
215	Cool 1 Flux Heater: Lo Deviation Alarm	55 OVEN 18/05/2015 12:24:25 Cool 1 Flux Heater: Lo Deviation Alarm 15°C	the temperature value (15 in the example) when the warning is raised, 0 otherwise
216	Cool 2 Flux Heater: Lo Deviation Alarm	55 OVEN 18/05/2015 12:24:25 Cool 2 Flux Heater: Lo Deviation Alarm 15°C	the temperature value (15 in the example) when the warning is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
217	Cool 1 Flux Heater: Hi Deviation Alarm	55 OVEN 18/05/2015 12:24:25 Cool 1 Flux Heater: Hi Deviation Alarm 15°C	the temperature value (15 in the example) when the warning is raised, 0 otherwise
218	Cool 2 Flux Heater: Hi Deviation Alarm	55 OVEN 18/05/2015 12:24:25 Cool 2 Flux Heater: Hi Deviation Alarm 15°C	the temperature value (15 in the example) when the warning is raised, 0 otherwise
219	Cool 1 Flux Heater: Rise Rate Alarm on Channel	218 OVEN 12/12/2015 10:53:15 Cool 1 Flux Heater: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
220	Cool 2 Flux Heater: Rise Rate Alarm on Channel	218 OVEN 12/12/2015 10:53:15 Cool 2 Flux Heater: Rise Rate Alarm on Channel	1 when the event is raised, 0 otherwise
221	PPM Level out of warning band	18 OVEN 31/10/2014 08:43:06 PPM Level out of warning band	1 when the event is raised, 0 otherwise
222	Flux Burn off cycle activated	60 OVEN 27/03/2015 11:51:21 Flux Burn off cycle activated.	1 when the event is raised, 0 otherwise
223	Communication lost	6 OVEN 06/11/2013 20:53:45 Communications with the oven was lost!	1 when the event is raised, 0 otherwise
224	Rail is not moving	80 OVEN 12/05/2015 19:09:02 Rail 1 Adjustment: The rail is not moving, check for failure	1 when the event is raised, 0 otherwise
225	Redundant OverTemperature Alarm	6 OVEN 06/11/2013 20:53:45 Redundant OverTemperature Alarm	1 when the event is raised, 0 otherwise
226	Board Drop lane	218 OVEN 12/12/2015 10:53:15 Board Drop lane	1 when the event is raised, 0 otherwise
227	Dansensor: connection up	38 OVEN 17/11/2014 01:49:38 Dansensor connection up	1 when the event is raised, 0 otherwise
228	Dansensor: the connection broken	58 OVEN 04/04/2016 09:13:28 The connection to the PBI Dansensor software has been broken!	1 when the event is raised, 0 otherwise
229	Dansensor: connection not established	25 OVEN 11/12/2014 14:02:23 The connection to the PBI Dansensor	1 when the event is raised, 0 otherwise

DLT input position	Value type	Event row example (*)	Mapping required action
		software has not been established!	
230	Dansensor: the level for alarm 1 has been sent	11 OVEN 23/11/2014 06:13:46 The level for alarm 1 has been sent to the Dansensor software 0PPM	1 when the event is raised, 0 otherwise
231	Dansensor: an attempt to set the level for alarm 1 failed	11 OVEN 23/11/2014 06:13:46 An attempt to set the PPM in the Dansensor device failed	1 when the event is raised, 0 otherwise
232	Dansensor: the standby alarm1 level was sent	68 OVEN 14/09/2014 15:22:51 The standby oxygen PPM was sent to the Dansensor software	1 when the event is raised, 0 otherwise
233	Dansensor: the oxygen ppm has been sent	68 OVEN 14/09/2014 15:22:51 The oxygen ppm has been sent to the Dansensor software 0PPM	1 when the event is raised, 0 otherwise
234	Dansensor: an attempt to set the PPM failed	25 OVEN 11/12/2014 14:02:23 An attempt to set the level for alarm 1 in the Dansensor failed	1 when the event is raised, 0 otherwise
235	Dansensor: the standby oxygen PPM was sent	25 OVEN 11/12/2014 14:02:23 The standby alarm1 level was sent to the Dansensor software	1 when the event is raised, 0 otherwise

(*) These examples may not have always a direct correspondence in the historical data set reported by BSL.

12 List of Figures and Tables

12.1 Figures

12.2 Tables

13 References

D. P. Kingma, J. B. (2015, December 22). Adam: A Method for Stochastic Optimization. *3rd International Conference for Learning Representations.* San Diego.

Entropy, C. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Cross_entropy

Frank, R. (1962). *Principles of Neurodynamics.* Washington: Spartan Books. Retrieved from https://en.wikipedia.org/wiki/Feedforward_neural_network

Gas dataset. (n.d.). Retrieved from https://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+exposed+to+turbulent+gas+mixtures

Haykin, S. (2009). *Neural Networks and Learning Machines.* Hamilton, Ontario, Canada: McMaster University.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation. 9 (8): 1735–1780.*

Hyperopt. (n.d.). *Hyperopt*. Retrieved from https://hyperopt.github.io/hyperopt/

Karpathyc, A. (2015, May 21). *Andrej Karpathyc blog*. Retrieved from rnn-effectiveness: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

London metal dataset. (n.d.). Retrieved from https://www.quandl.com/data/LME-London-Metal-Exchange

Mitchell, T. (1997). *Machine Learning.* McGraw-Hill International Editions.

P. Doetsch, A. Z. (2016). *RETURNN: The RWTH Extensible Training framework for Universal Recurrent Neural Networksc.*

RETURNN. (n.d.). *TensorFlow LSTM benchmark*. Retrieved from returnn.readthedocs.io: https://returnn.readthedocs.io/en/latest/tf_lstm_benchmark.html

Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review, 65, 386-408.*

S. J. Reddi, S. K. (2018). On the Convergence of Adam and Beyond. *Internation Conference on Learning Representations.*

S. van der Walt, S. C. (2011). The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering, vol. 13, no. 2, pp. 22-30, 13*(2), 22-30.

Scipy reference. (n.d.). Retrieved from https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.save.html

Wikipedia. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Artificial_neural_network

Wikipedia. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Recurrent_neural_network#Fully_recurrent

Wikipedia. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Vanishing_gradient_problem

Wikipedia. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Long_short-term_memory

Wikipedia. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Confusion_matrix

Wikipedia. (n.d.). *Wikipedia*. Retrieved from Positive and negative predictive values: https://en.wikipedia.org/wiki/Positive_and_negative_predictive_values

Xu-Ying Liu, J. W.-H. (2009). Exploratory Undersampling for Class-Imbalance Learning. *IEEE Trans. Syst., Man, Cybern. B, Appl. Rev,.*