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

D5.3 Continuous Deep Learning Toolkit for Real Time Adaptation I

Date: 2018-01-05

Version 1.1

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: D5.3 Continuous deep learning toolkit for real time adaptation I v1.1.docx
Document version: 1.1
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: R

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	04-10-2017	Definition of table of contents
0.2	Luca Boulard	30-10-2017	Reporting of first experiments on synthetic data
0.3	Luca Boulard	05-11-2017	Framework comparison
0.4	Nadir Raimondo	30-11-2017	Reporting of latest experiments on real data
0.5	Paolo Vergori	07-11-2017	Introduction and data assessment
0.6	Nadir Raimondo	13-11-2017	Chapter 6 refinement and introduction. Conclusions
0.9	Paolo Vergori	19-12-2017	Final version for internal peer reviewer #1
1.0	Paolo Vergori	22-12-2017	Final version for review peer reviewer #2
1.1	Paolo Vergori	05-01-2018	Submitted to EC

Internal review history:

Reviewed by	Date	Summary of comments
Diaz Rodriguez, Rodrigo (ATOS)	2017-12-21	The document is well written and structured, providing enough details of the work done in this activity. I think it should be defined, in this document or in the next one, a detailed plan for mitigating risks.
Willie Lawton (TNI-UCC)	2018-01-04	In general it looks a very good deliverable.

Contents

1 Executive Summary

The present document named “D5.3 Continuous deep learning toolkit for real time adaptation I v1.1” 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 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 version 1.1, submitted at M16, are highlight the results of the first iteration of project’s 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 are provided for each of the addressed project’s use cases. A second iteration of this document, named “D5.4 Continuous deep learning toolkit for real time adaptation II” will be submitted at M30 and will update the results of task “5.2 – Continuous Deep Learning Toolkit for real time adaptation” through an iterative approach.

2 Abbreviations and acronyms

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
TRL	Technology Readiness Level

3 Introduction

3.1 Summary

In this document it is described the development carried out in COMPOSITION's project task 5.2, named Continuous Learning Toolkit for Real Time Adaptation.

The document is structured in seven 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. In specific, at first results on extensive tests on synthetic data representing multiple possible scenarios are going to be reported, followed up by real data from one of the end users. At last, in the final section, a comprehensive analysis of the achieved results is presented. Furthermore, a detailed report of required work that will be necessary to complete this activity and tackle future challenges is included in the final section.

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

3.2 Background

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

As algorithms are progressing 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 real power over Artificial Neural Networks (ANNs) will not be achieved by withholding intellectual proprieties over algorithms or frameworks, that in fact are released open source and progressively updated by the community, but this 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 historical data availability. As it will be 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 focus for this first deliverable has been put 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.

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 arise at the intersection of a number of different 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 [1] and dating batch 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 improve the model performance until it can be deployed and applied to predict the content of new unseen and unlabelled images.

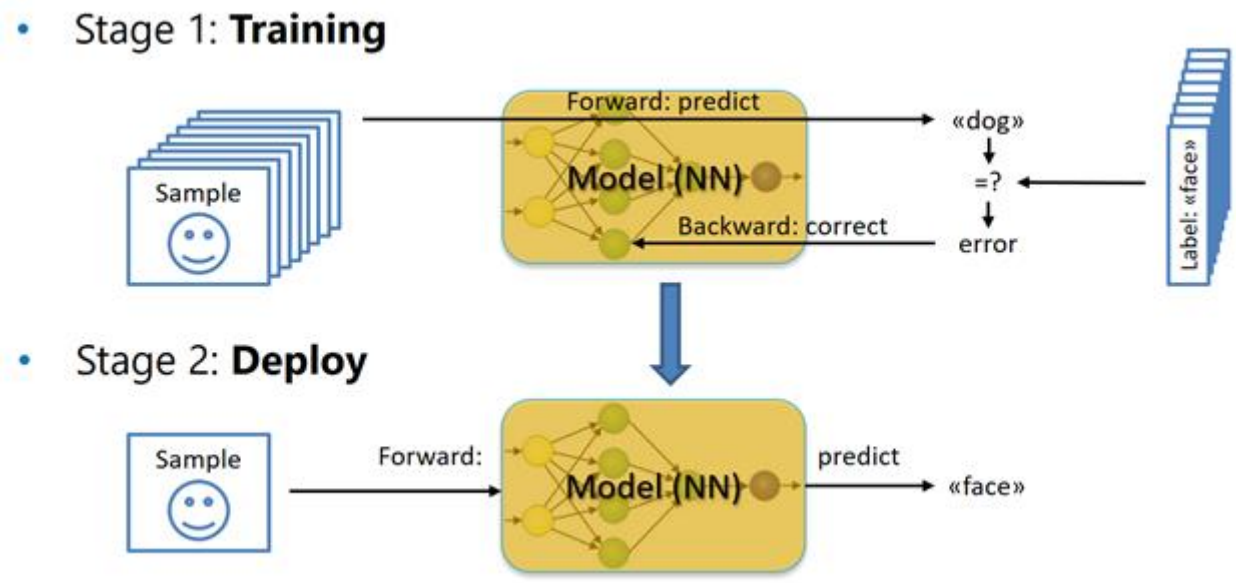

Figure 1: example of machine learning stages for the task of image content prediction

ML is therefore a good approach in a plethora of different situations and 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 [2] as a computing systems 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 that 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 collections 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 a Frank Rosenblatt machine-learning algorithm named “Perceptron” [3]. Two kind of perceptron have been generated as described in [2] [4]:

- 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 of 3 neurons each and one output layer. All the connections are between adjacent layers and neurons within a single layer share no connections.

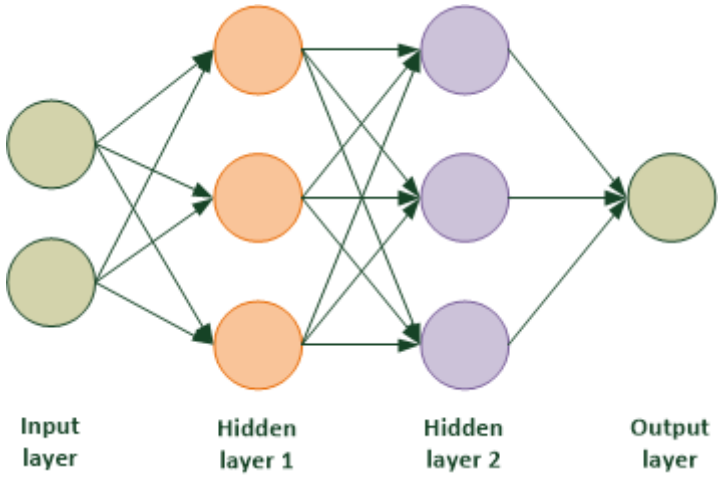

Figure 2: network layers

4.4 Recurrent neural network and long-short term memory network

The excerpt from literature continues in [5], where all this information is widely available. A brief and concise dissertation continues in the followings.

A Recurrent Neural Network (RNN), as defined in [5], 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 move better guesses. As described in [6], 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 [7] was introduced by Hochreiter and Schmidhuber in 1997 [8] 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 often 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 not unlikely 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, allowing for simplified API and efficient code reuse. Some of them come from academia while other 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 to fulfil and task to address.

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

As part of task 5.2 we conducted an extensive review 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 algorithm and have more modern overall architecture. In any case, actively developed frameworks are to be preferred.
- Usage licence: In COMPOSITION, we are interested 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 a detailed and up to date documentation is an important consideration for ML framework adoption as for any other piece of software. The same applies for dimension of the adopting community, which correlates to the availability of tutorials examples and blog posts.
- 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 GPU computing), 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, 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 it to work in 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 with respect to C/C++ so that such APIs bring a consistent speedup in model design. 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 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 most 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 incredibly computationally eager, it is possible to contain the execution time by multithreading: the computation workload is delegated to multicore hardware such as CPUs where is 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 and 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).

In Table 1 to Table 5 compares the frameworks under different features:

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
H_2O	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 1: comparison of frameworks - creator, first and last releases

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
H_2O	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 2: comparison of frameworks – licensing

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)
H_2O	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 3: comparison of frameworks - supported platforms & languages

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)
H_2O	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 4: comparison of frameworks - GPU & distributed computing

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
H_2O	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

Table 5: comparison of frameworks - algorithm support

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 over the popular platform Stack Overflow along several years. Only few frameworks are considered.

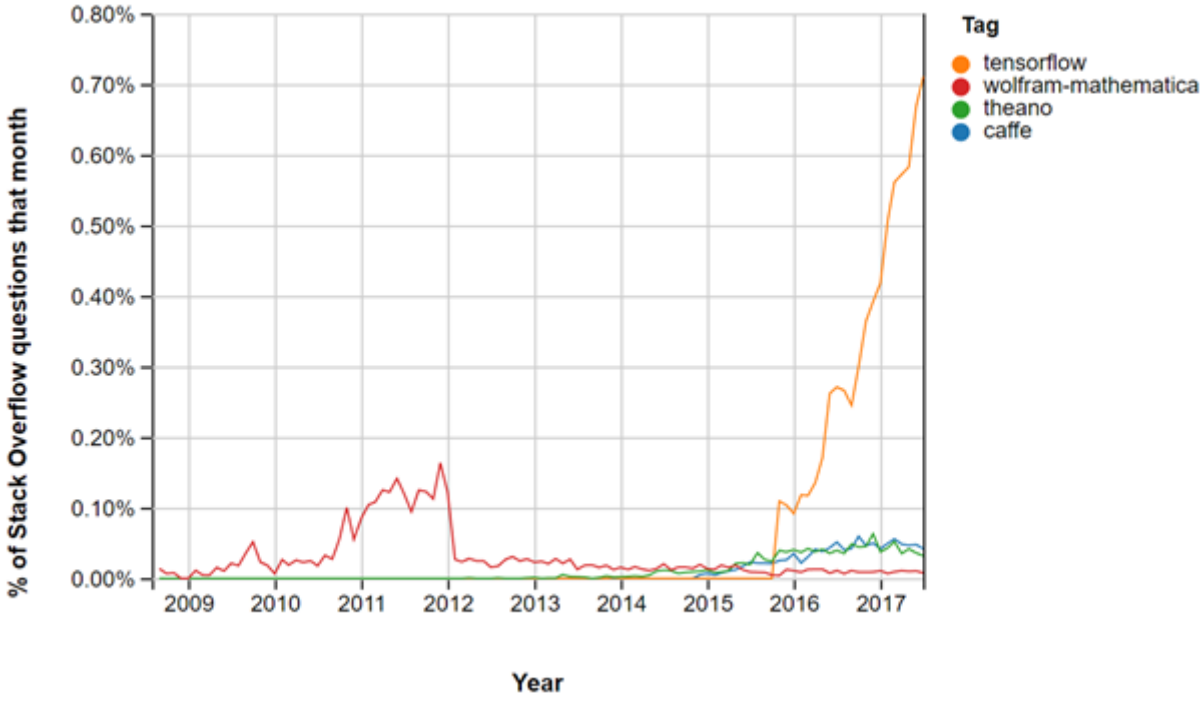

Figure 3: trends of Stack Overflow questions

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.

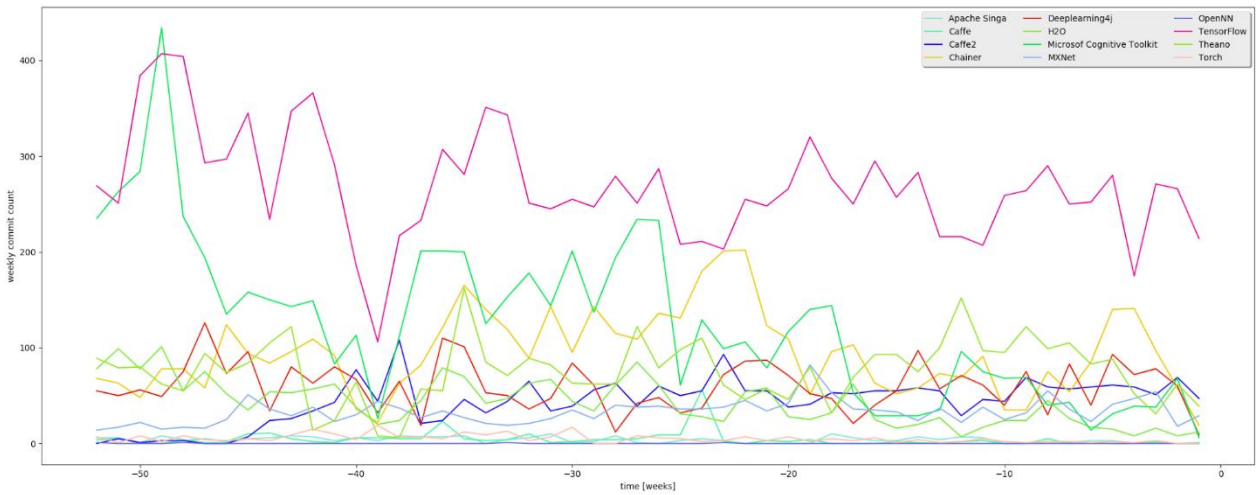

Figure 4: trends of weekly GitHub commits across the last year

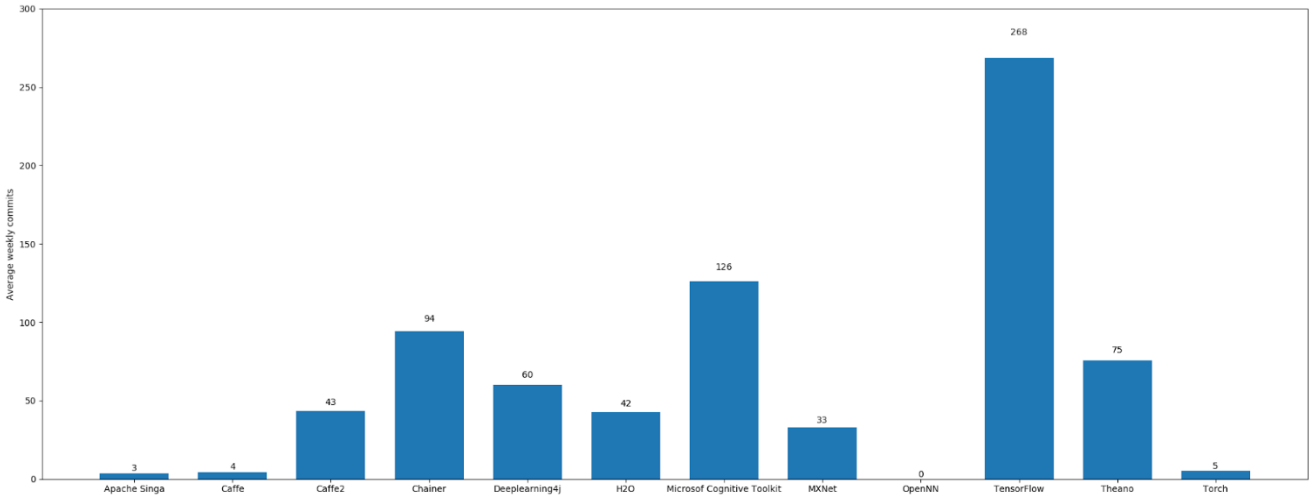

Figure 5: average numbers of weekly GitHub commits in the last year

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.

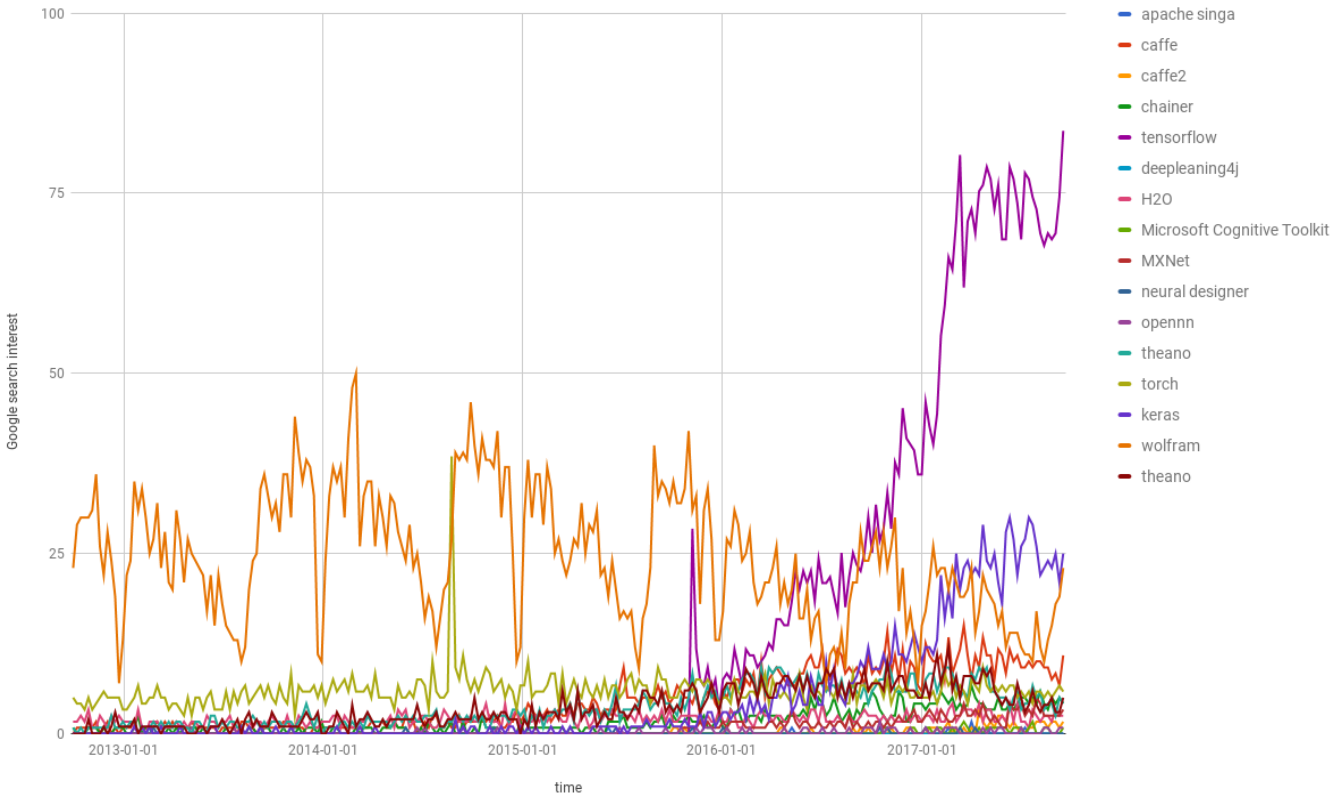

Figure 6: Google search trends

Some considerations emerged from the data reported so far.

The majority of deep learning frameworks is 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 hands, 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, discourage from 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 were individuated as in Table 6 to support the final framework choice. Anyway, it is worth noting that the framework evolution is very fast and new features are incorporated every few months so that the trade-off of pros and cons of each solution are likely to be changed by time writing.

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.	Most popular for Computer Vision tasks and CNN.

		Not extensible, may be difficult to apply outside Computer Vision.	
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 framework 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.

Table 6: pros and cons of open source frameworks

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, to its APIs operating different abstraction level allowing to trade-off between ease of coding and control of algorithm detail. Being developed and adopted by Google for its AI projects seemed to offer good prospects over community width, continuous development, quality of the documentation, efficiency of deployment systems.

In addition, H_2O, 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.

When, in a second time as clearly stated in future sections 6 and 7.4, the need for Recurrent Neural Network clearly emerged, a new evaluation was carried on. By the time, TensorFlow passed from version 0.8 to version 1.3, adding more features and improvements under many aspects, including RNN support, so that it seemed inconvenient to switch to a very different framework such as Microsoft Cognitive Toolkit (which has gone through major improvements and seems now to be a very convenient, flexible and fast library too).

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. These datasets are mandatory for a component such as the Deep Learning Toolkit whose core is mainly composed of deep artificial neural network models.

In fact, in order to achieve a 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 use case scenario. Once deployed, it will process live data streams provided by the Big Data Analytics component, producing meaningful predictions and updating them 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 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 list 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 in 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:

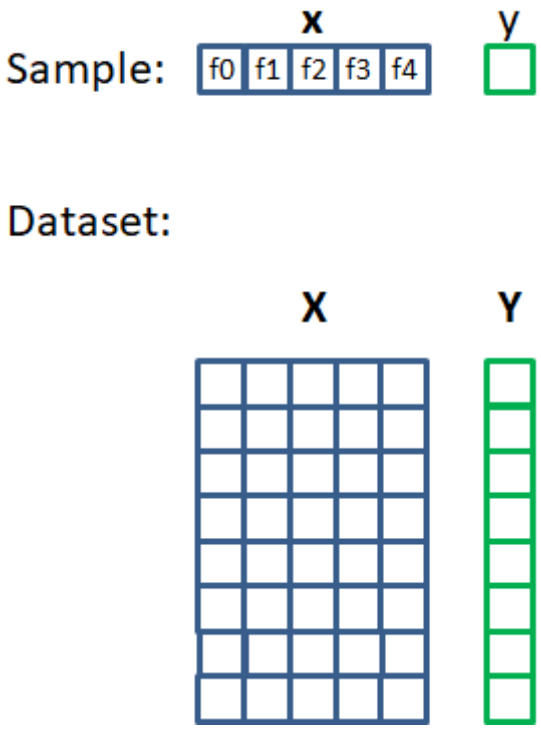

Figure 7: dataset structure

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. In the following sections is 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:
    - Predictive maintenance (UC-BSL-2)
    - Maintenance Decision Support (UC-KLE-1)
    - Scrap metal / Fill bin (UC-KLE-3, UC-KLE-4, UC-ELDIA-1, UC-ELDIA-2)
    - Prices and Collection (UC-KLE-4, UC-KLE-5, UC-KLE-6)

In the (hopefully) unlikely event where an historical dataset is missing and therefore the action of performing the initial training would not be possible, 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. Nevertheless, it is demonstrated in literature by the ANNs guru Simon Haykin in its biblical manual Neural Networks and Learning Machines [9] on page 187 where he states that back propagation always converges, although the rate can be slow and in its own words: it "can be excruciating". This is also proved in chapter 7 by the experiments on the synthetic data. It is therefore safe to assume that a good level of accuracy, and therefore convergence, is reachable in a finite time by using specific topologies and models of artificial neural networks.

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 required to address, 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 here, because 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 required to under sample the amount of data in the normal state, balancing the input for the model. Hence, when we talk about the category of classification problems, like the ones it is possible to reduce most of the intra-factory use cases, if we want to have 100K balanced samples, it is required to have $\frac{100K}{NumberOfClasses}$ for each class the model is required to identify.

6.1 Predictive maintenance (UC-BSL-2)

6.1.1 Background

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

6.1.2 Data Overview

BSL provided a large dataset related to four reflow ovens (Brady, Tachy, Rhythmia and NMD). For each oven, the dataset encompasses data files, one per day, covering the period 2008-2017. Actually, the beginning of the recording, 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 differ 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 log 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, we 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.

Figure 8 below show some plots of the Brady oven blowers corresponding to faults. The color meaning is the following:

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

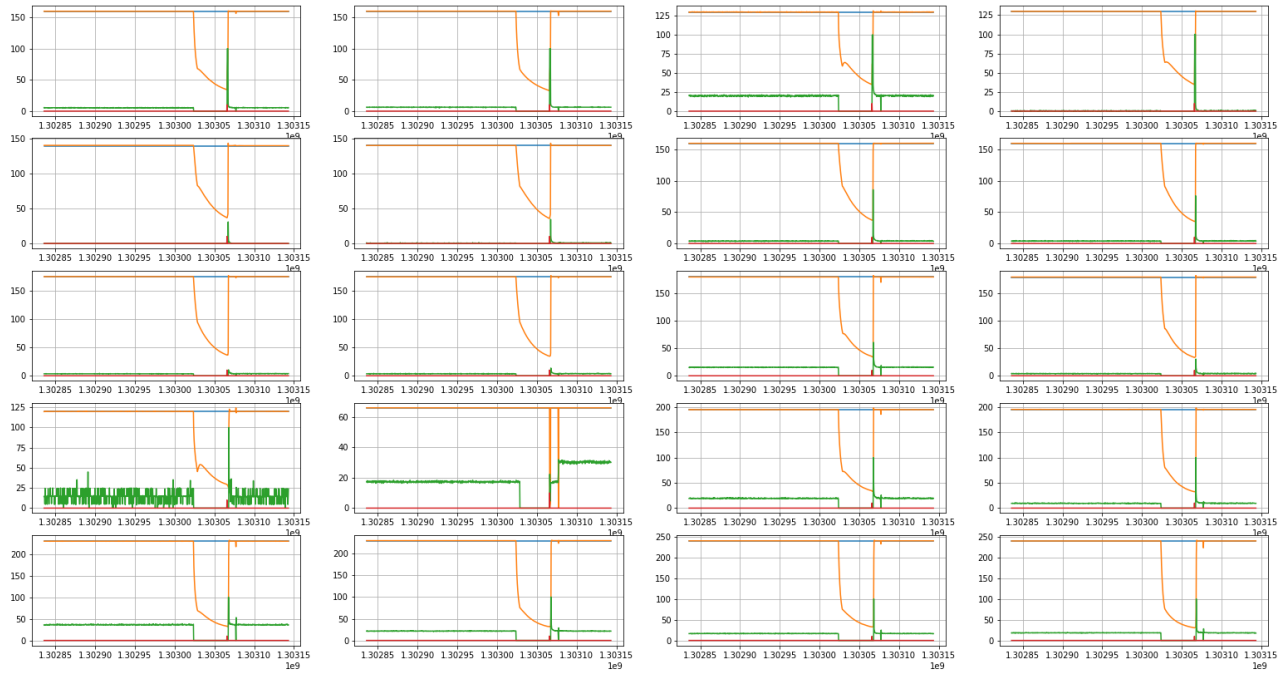

Figure 8: Brady oven dataset from 04/15/2011 to 04/18/2011

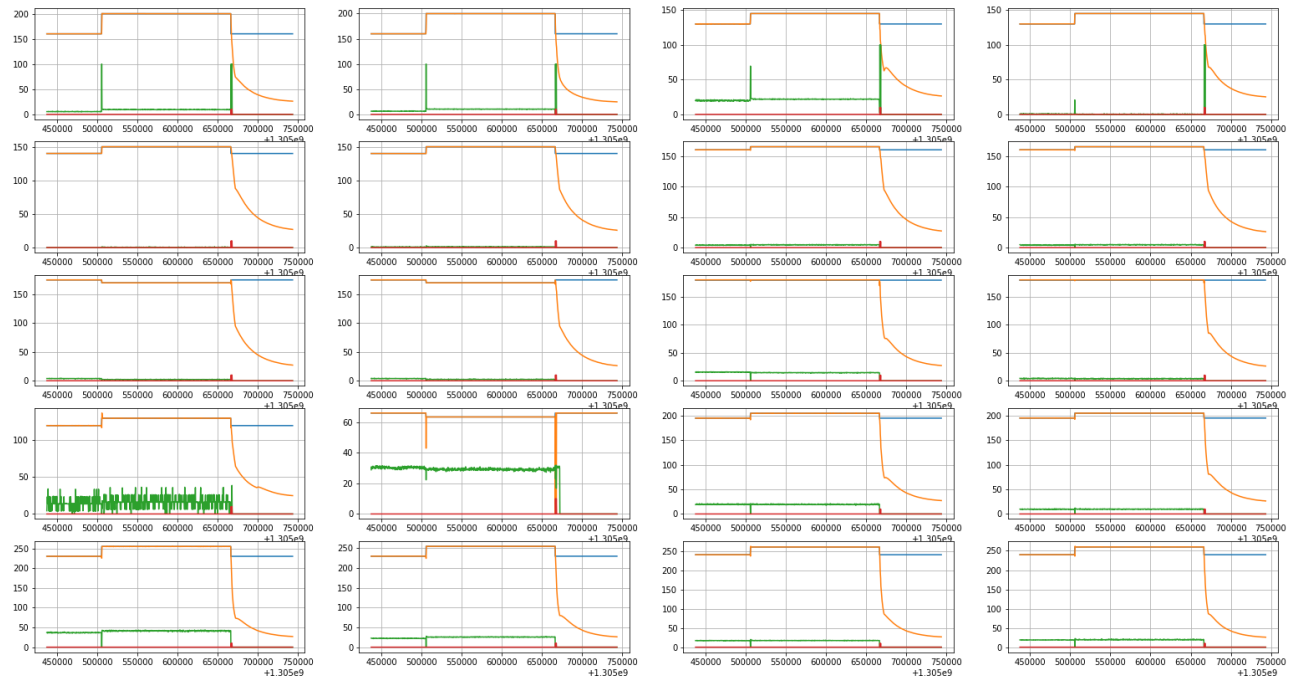

Figure 9: Brady oven dataset from 05/15/2011 to 05/18/2011

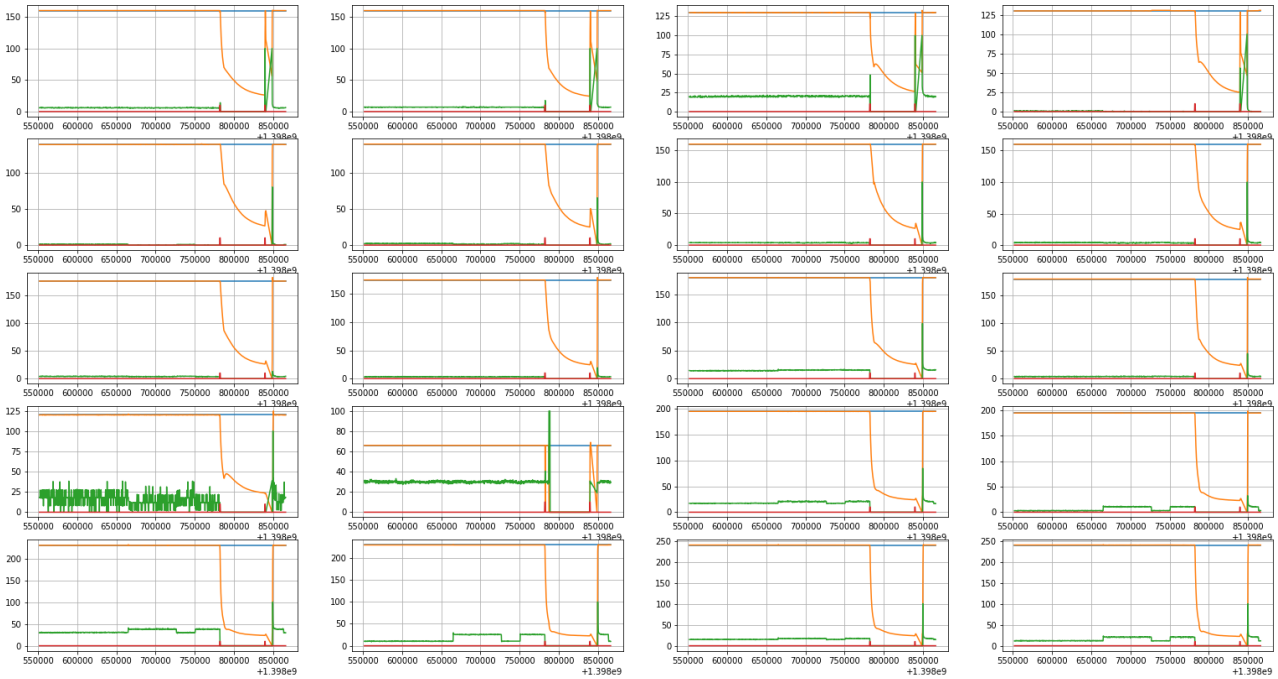

Figure 10: Brady oven dataset from 04/27/2014 to 04/30/2014

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.

6.1.3 Fitness for usage

The dataset focus and size is suitable for 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 too high and the single samples become irrelevant and results in an unbalanced situation in which 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 measures of tens blowers, the failure events do not provide any information for identifying the damaged blower. This may affect the capability to correctly predict future failures.

6.1.4 Data assessment

Given the provided data, 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 provide 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.

6.2 Maintenance Decision Support (UC-KLE-1)

6.2.1 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.

6.2.2 Data Overview

Kleemann provided a historical dataset of failures featuring:

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

6.2.3 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.

6.2.4 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 Klemann, 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.

6.3 Fill level and classification use cases (UC-KLE-3 UC-KLE-4 UC-ELDIA-1 UC-ELDIA-2)

6.3.1 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.

6.3.2 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.

6.3.3 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 (Klemann 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.

6.3.4 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.

6.4 Prices and logistics (UC-KLE-4, UC-KLE-5, UC-KLE-6)

6.4.1 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 at what 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 formed 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.

6.4.2 Data Overview

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

Klemann 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 summarize 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 month 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 month 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 provide all waste categories, the dataset has a relevant amount of missing data.

6.4.3 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 wider 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.

6.4.4 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, these 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.

In the next iteration of this document, it will be possible to see a leap forward in the applicability of the Deep Learning Toolkit in this use case, when enough data will be collected. Detailed action plan on how this problem is going to be addressed is presented in chapter 8.

7 Deep Learning Toolkit for continuous learning design and testing

7.1 Introduction

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 H_2O 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 H_2O 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. Results are reported in section 7.2.

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 7.4 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 7.4.2. 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 briefly describing the lab scale deployment in a Docker container that has been provided as software output for this first delivery as a private image. 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.

7.2 Feed Forward NN in TensorFlow

7.2.1 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 H_2O 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.

7.2.1.1 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

7.2.1.2 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.

7.2.1.3 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.

7.2.2 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 [10].

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.

7.2.3 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 setup are described in Table 7.

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 specs	
• 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	

Table 7: CPU vs GPU setup

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.

7.2.3.1 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.

Table 8, 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.

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

Table 8: Low-level API - CPU vs GPU

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 [11] 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 beidentified.
- 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.

7.2.3.2 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 9: 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.

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

Table 9: High-level API - CPU vs GPU

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.

7.3 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.

7.3.1 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.

7.3.2 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 trend 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.

7.3.3 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 Klemann 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.

As shown in Figure 11 price trends along time (per scrap, per customer) follow an asymptotic growth to emulate inflation and is corrupted by additive uniform noise in range [-1,+1]€ as can be seen in Figure 12. 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.

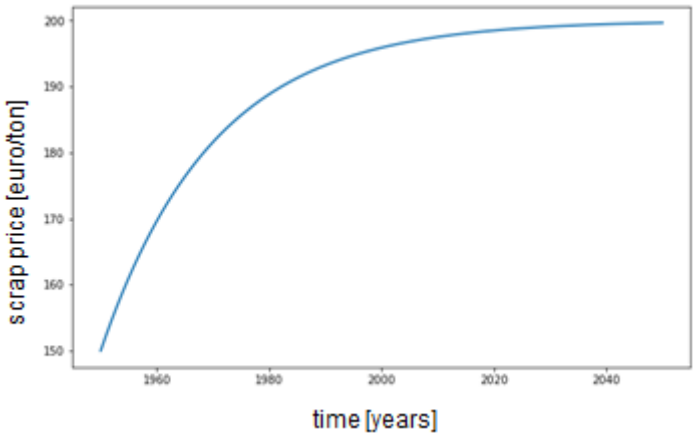

Figure 11: ground truth generating function of a single price trend

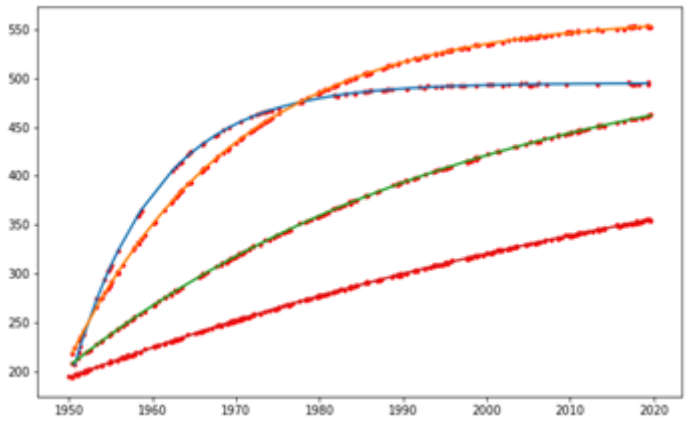

Figure 12: four different price trends: ground truth function and noise corrupted samples

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 13, and a continuous convergence of training and validation error.

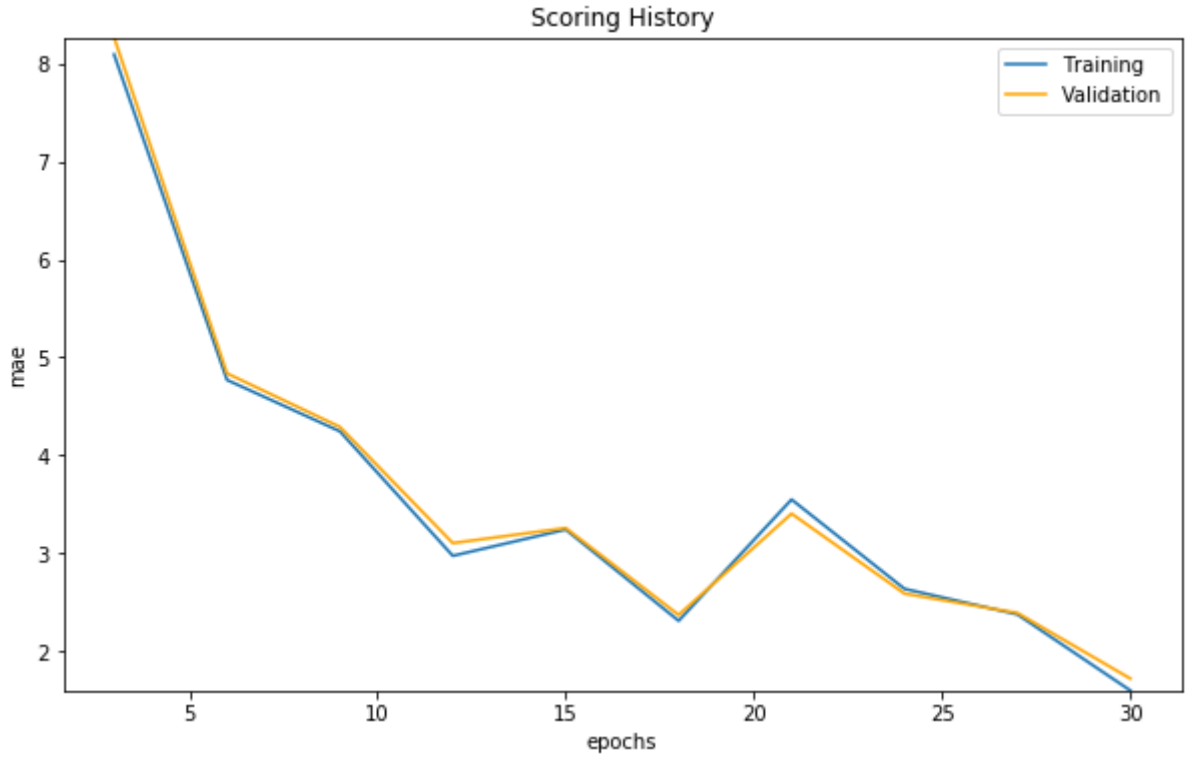

Figure 13: variation of mean absolute error (MAE) along training epochs

A number of final performance metrics are reported in

Table 10 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.

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

Table 10: final performances of the trained neural network

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.

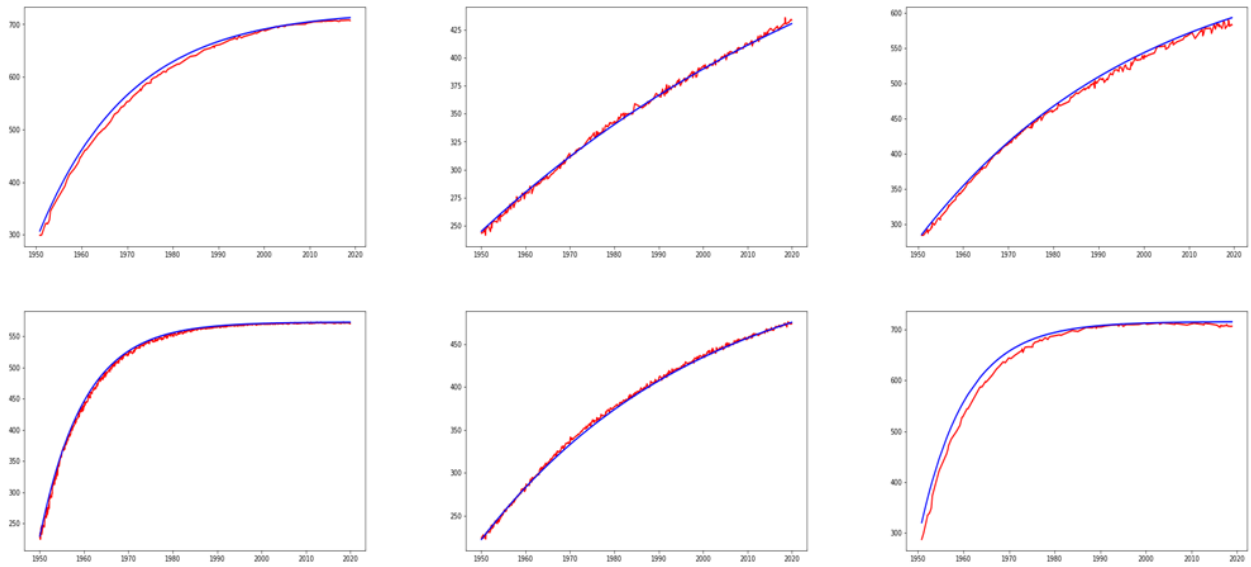

Figure 14: price trends - comparison of ground truth (blue) and predictions (red)

7.3.4 Updated demonstration on prices prediction

The first demonstration described in section 7.3.3 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 11.

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

Table 11: predictions' evaluation of future datasets

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.

7.4 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 kind 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.

7.4.1 LSTM regression on univariate time series

7.4.1.1 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 15, 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 follow:

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

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.

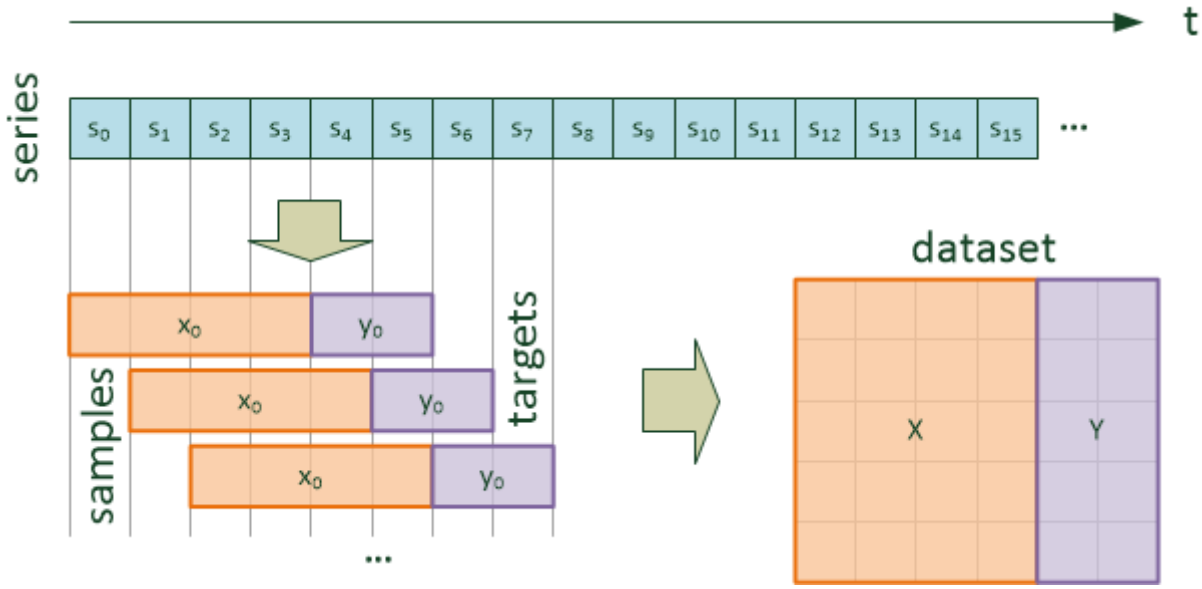

Figure 15: construction of a dataset from a time series (n_ts_is = 4, n_ts_out=2)

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.

7.4.1.2 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 16. 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.

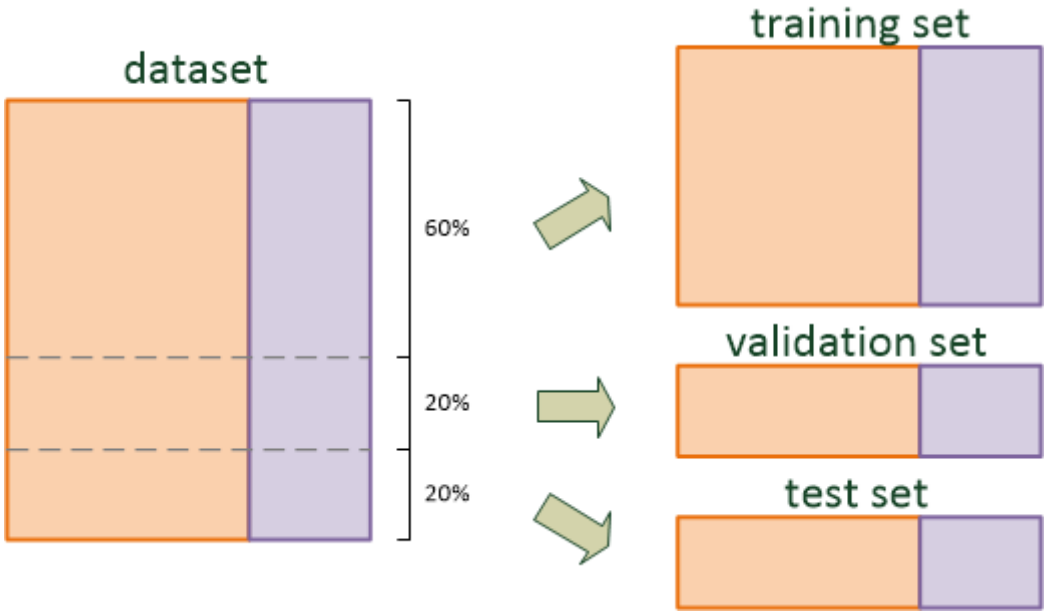

Figure 16: dataset splitting

7.4.1.3 LSTM network design, training and evaluation

In the figure below (Figure 17) 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.

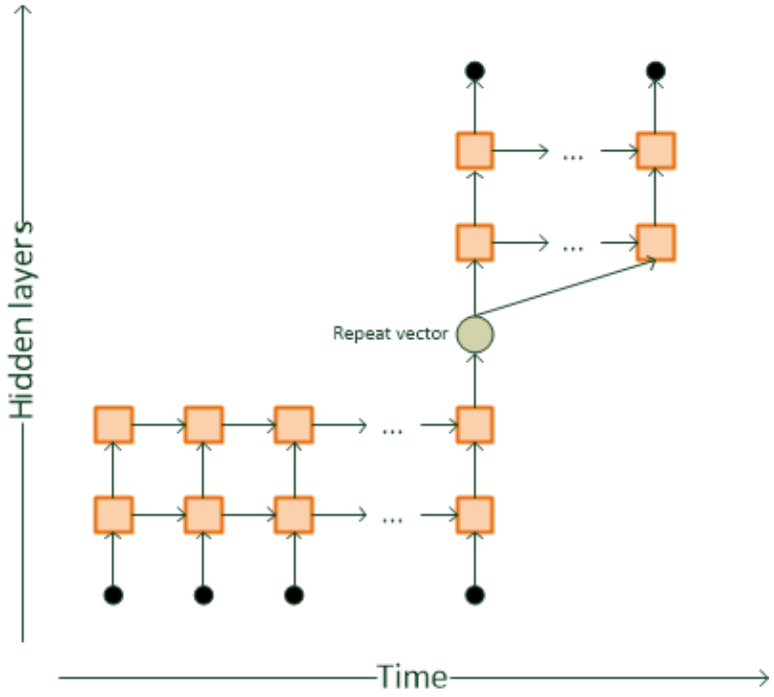

Figure 17: LSTM model structure

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 7.4.1.3.1 and 7.4.1.3.2, have been realized:

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

7.4.1.3.1 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 [12], have been considered to provide a much larger and more comprehensive data set. Figure 18 reports the selected interval of 1400 price samples from 2012 to 2017.

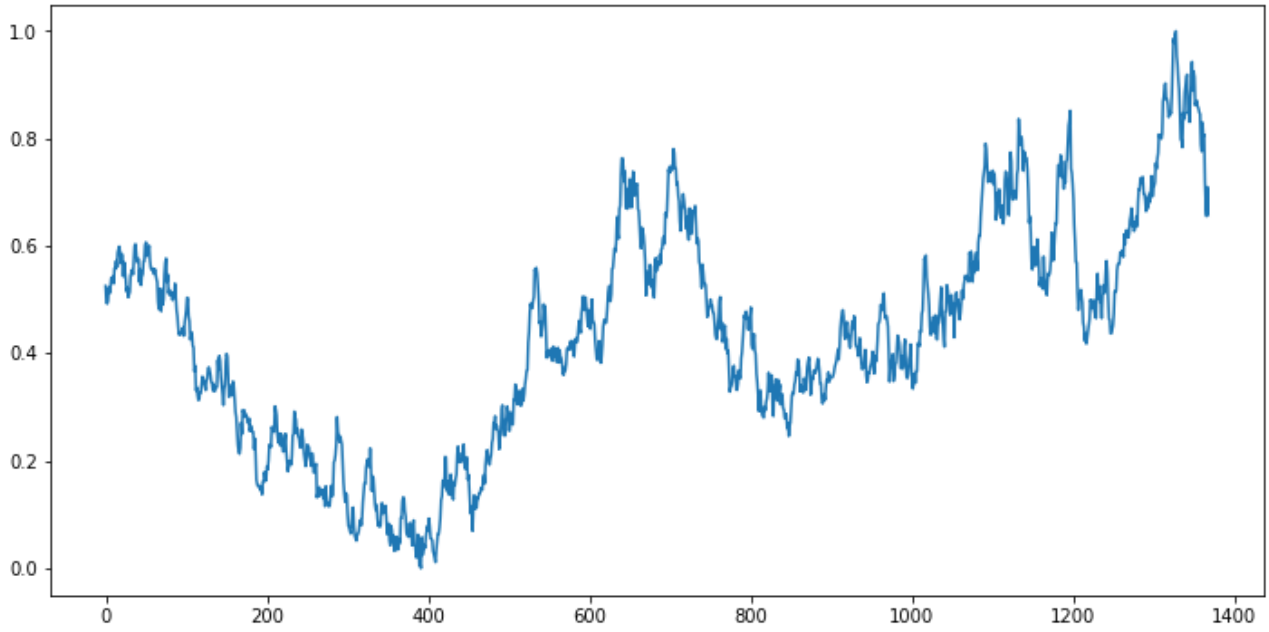

Figure 18: London Metal Exchange aluminium prices

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 19 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.

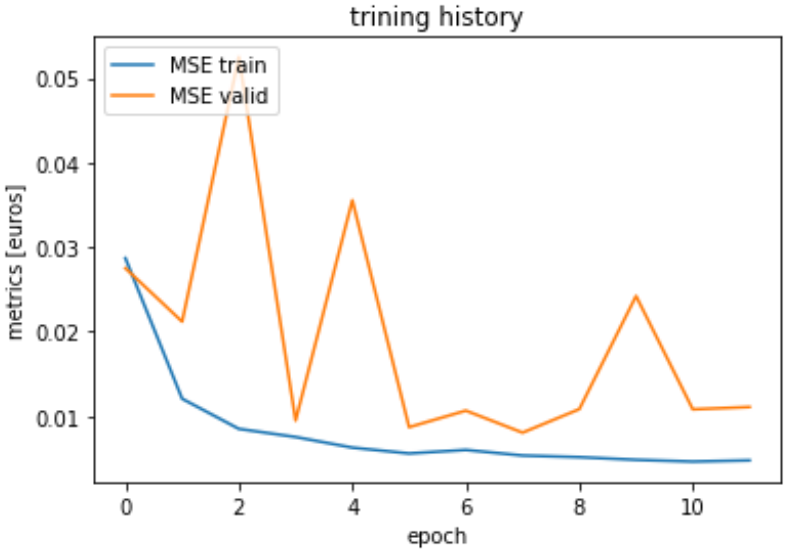

Figure 19: training results

The network can predict 15 time steps in the future with increasing predictions errors. In the Figure 20 and Figure 21 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.

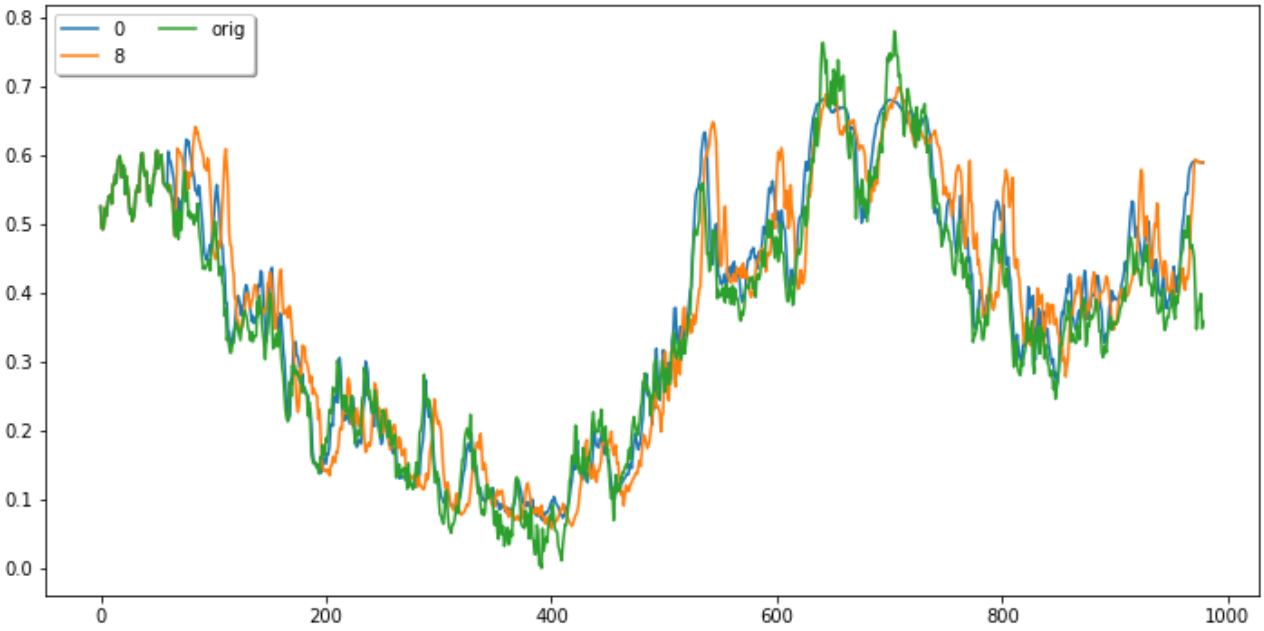

Figure 20: prediction results on training set

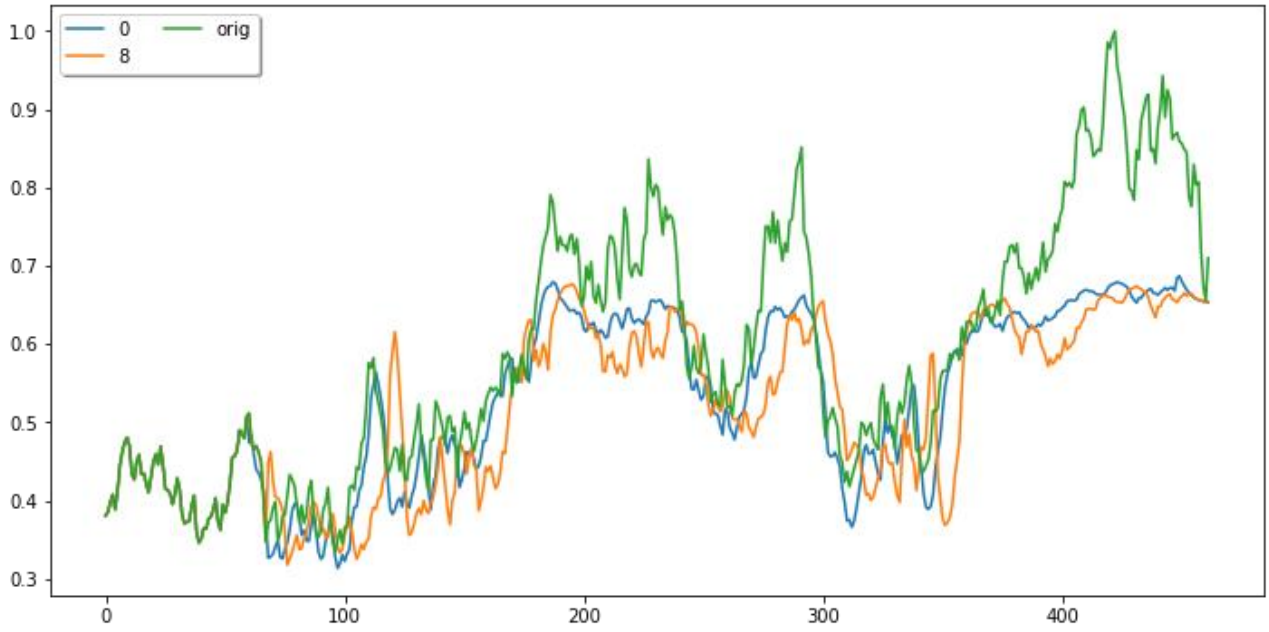

Figure 21: prediction result on validation set

As expected from the training results, the predictions' quality on the validation set (Figure 21) is less precise than the one produced with the verification of training set (Figure 20). A higher precision and therefore higher quality can be achieved by increasing the number of input samples.

7.4.1.3.2 Trigonometric series

In this experiment, a total of 4000 raw values have been obtained from the trigonometric function depicted in Figure 22.

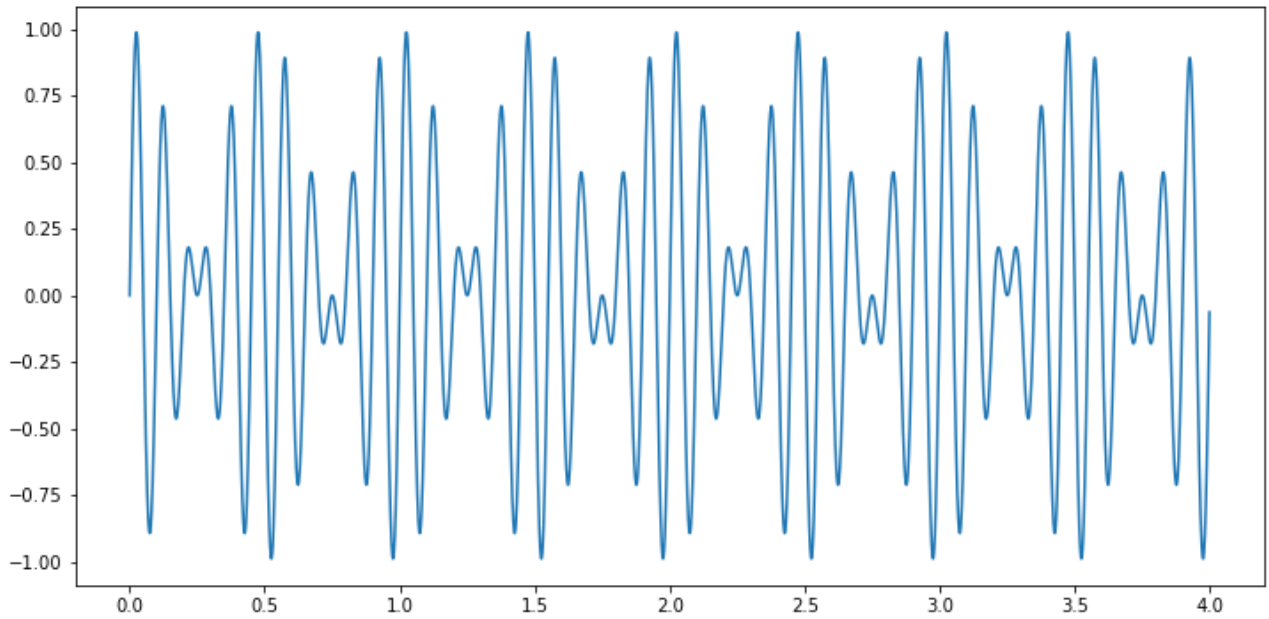

Figure 22: input time series

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 23 shows the training result.

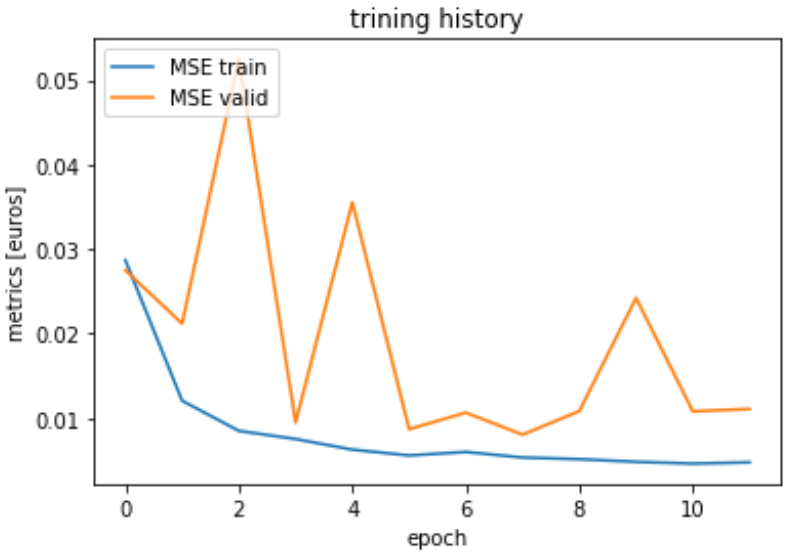

Figure 23: training results

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 24, 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.

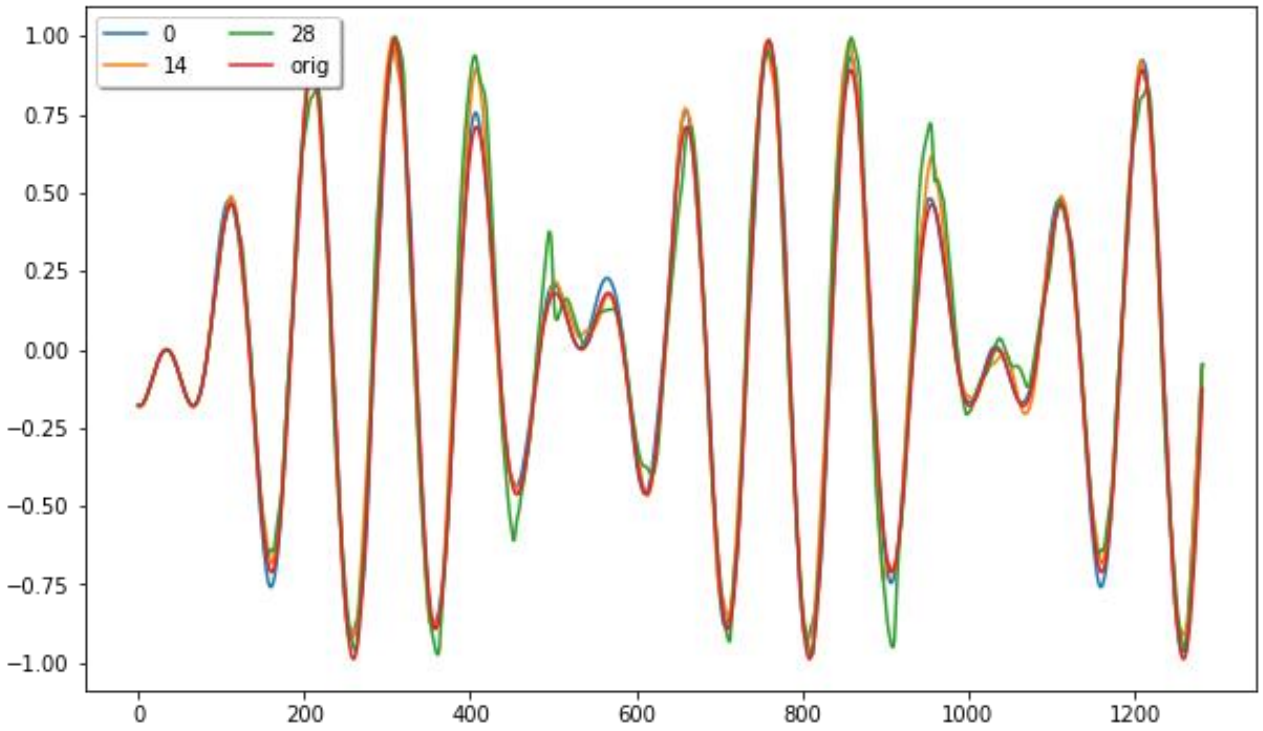

Figure 24: prediction results

7.4.2 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.

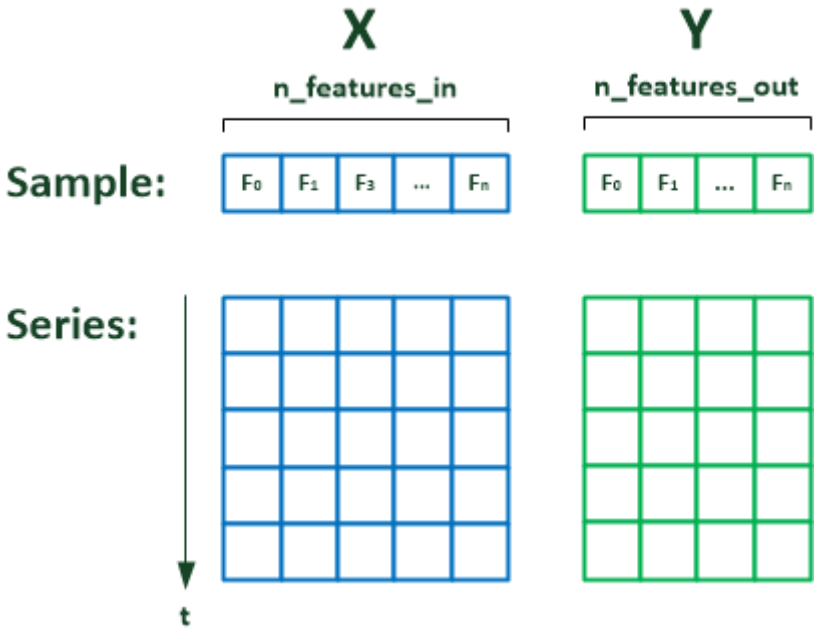

Figure 25: multivariate time series

As depicted in Figure 25, 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 7.4.1.1 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).

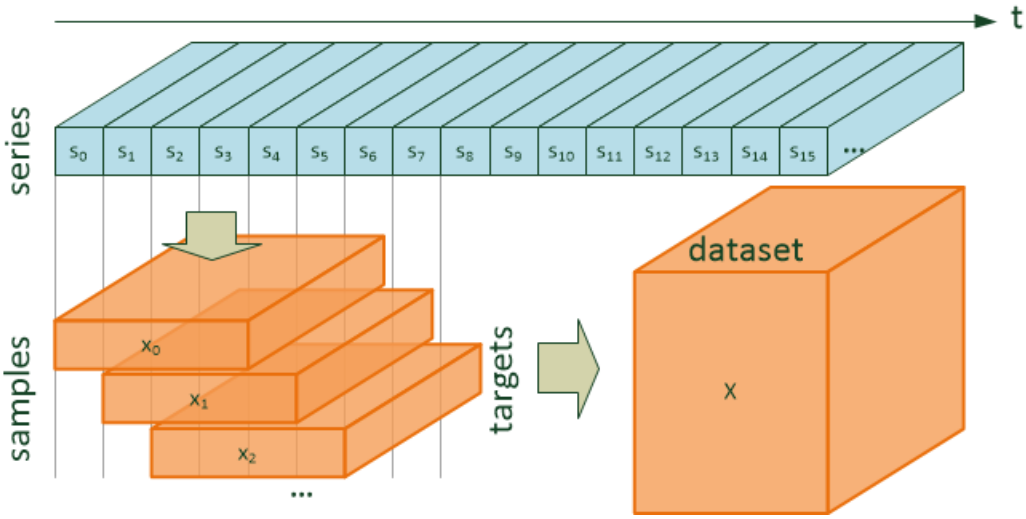

Figure 26: construction of a dataset from a time series (n_ts_is = 4)

In Figure 27 it is shown the resulting matrixes from the data formatting process on both X and Y vectors.

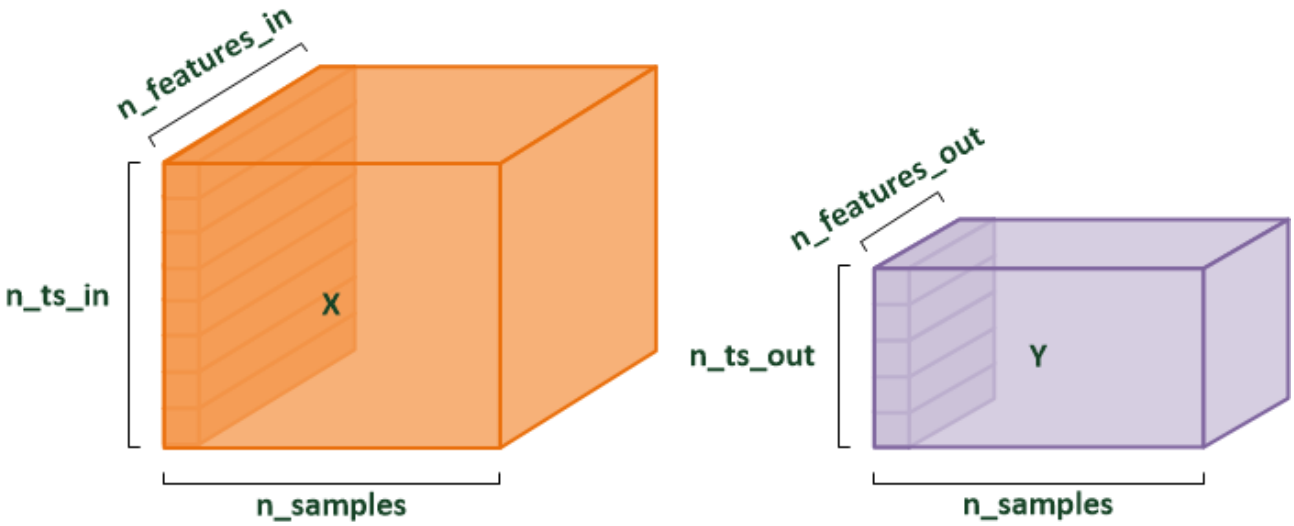

Figure 27: multivariate data set

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.

7.4.2.1 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 28, 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.

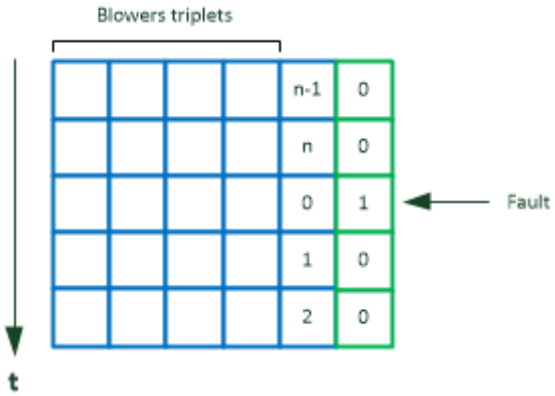

Figure 28: Training features

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 7.4.1.3), 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 26 shows the resulting matrixes structures used for all the experiments.

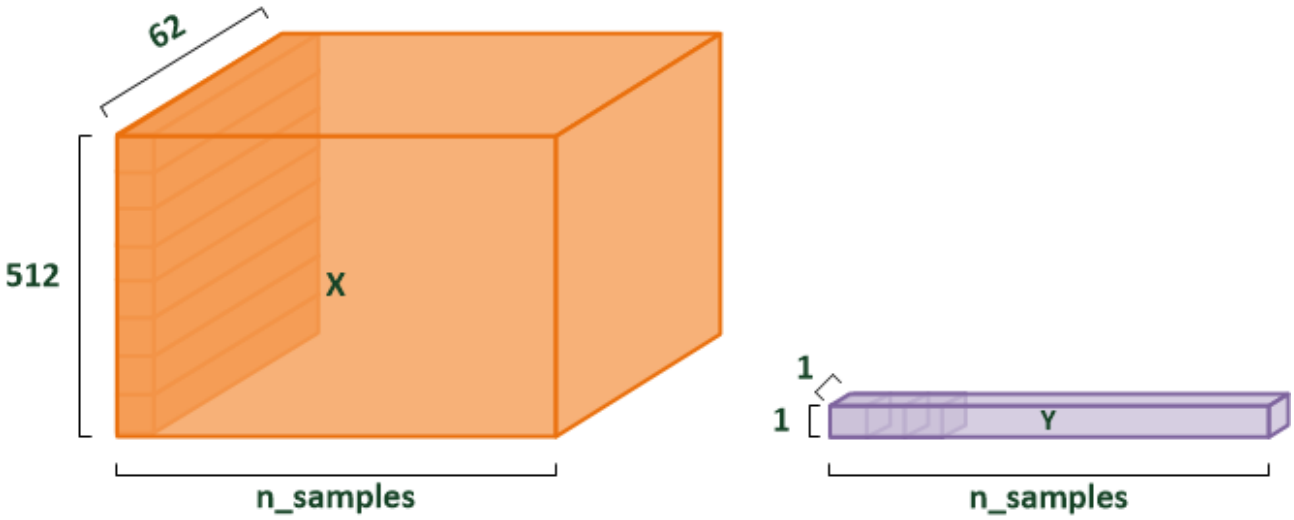

Figure 29: Input matrices for supervised learning test

7.4.2.1.1 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.

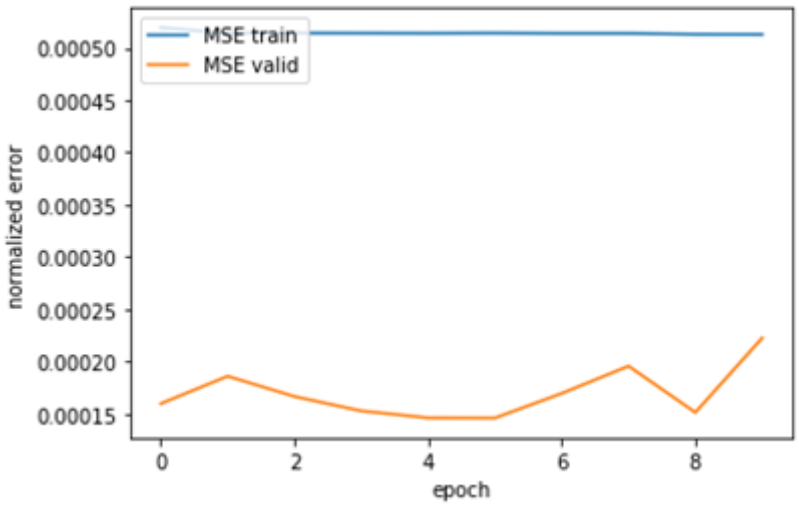

Figure 30: complete dataset training results

As depicted in Figure 30, 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.

7.4.2.1.2 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 [13]. 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 31, also the faults class has required some tweaks, in order to complement data with the necessary significance. Therefore, temporally close faults, represented by red dots in Figure 31: Event aggregation and selection. 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 31: Event aggregation and selection

Dataset have been then obtained with the very same process described in the previous experiment in section 7.4.2.1, 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 32.

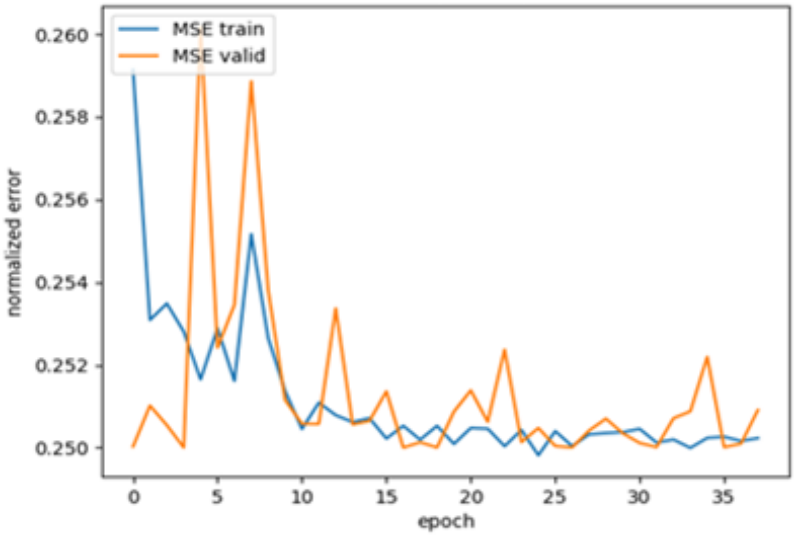

Figure 32: training results for aggregated dataset

As a clear results 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.

7.4.2.1.3 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.

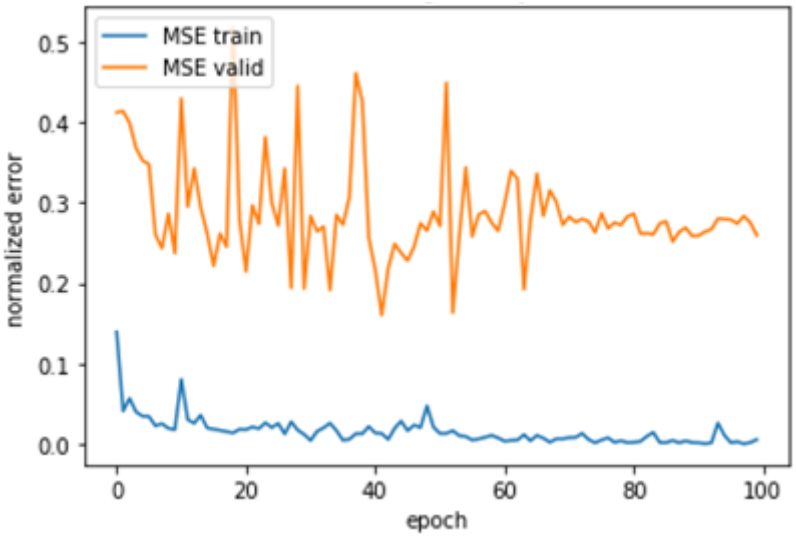

Figure 33: training results for aggregated and normalized dataset

In Figure 33 are shown the results. 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 described in section 7.4.2. 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.

7.4.2.2 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 0. 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 0 as a list of 512 samples with 62 features as a binary Numpy arrays. For a complete data format description please refer to [14]). 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.

7.4.3 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.

7.4.3.1 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 f_{max} is determined, the sampling frequency can be chosen to be $f_{sam} = 2\ f_{max}$.

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.

7.4.3.2 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 experiments description.

7.4.3.3 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.

7.4.3.4 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.

7.4.3.4.1 Test 0.0

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

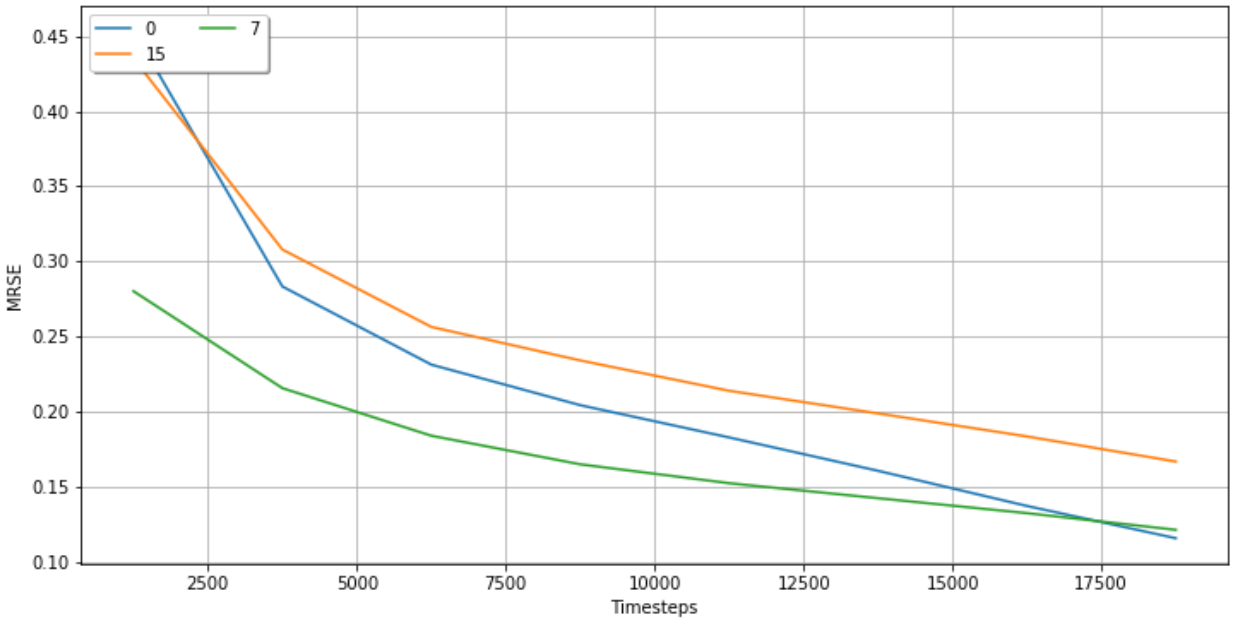

Figure 34: RMSE versus time steps

7.4.3.4.2 Test 0.1

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

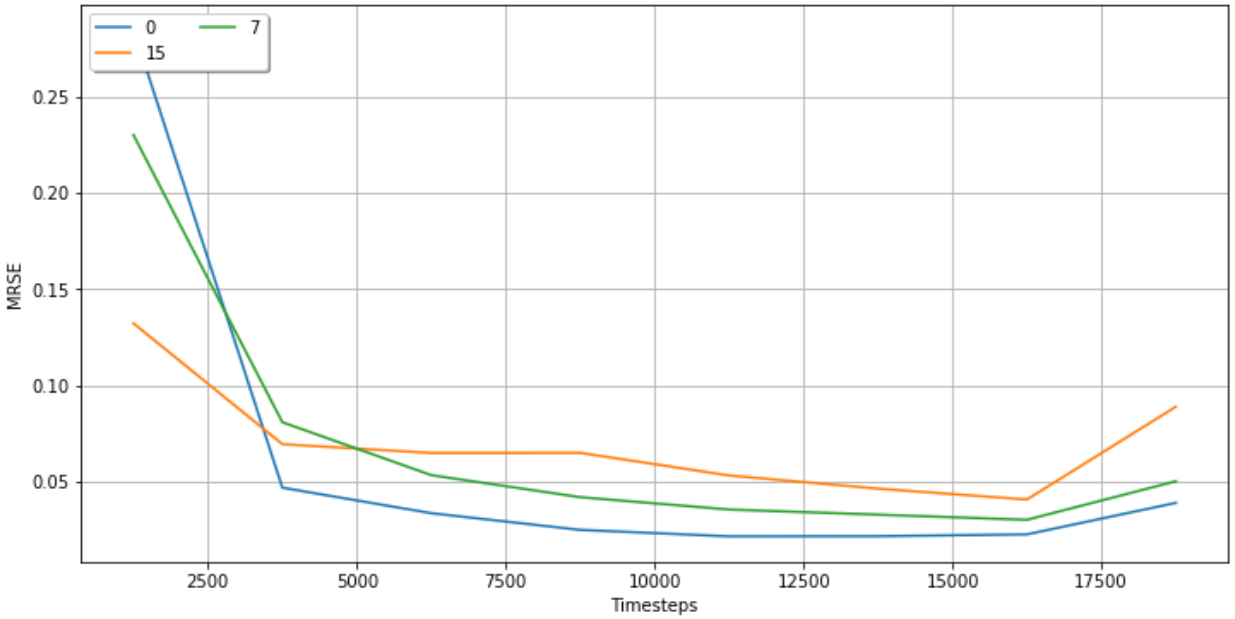

Figure 35: RMSE versus time steps

7.4.3.4.3 Test 0.2

- Initial learning rate: 0.01

- Training epochs per batch: 100

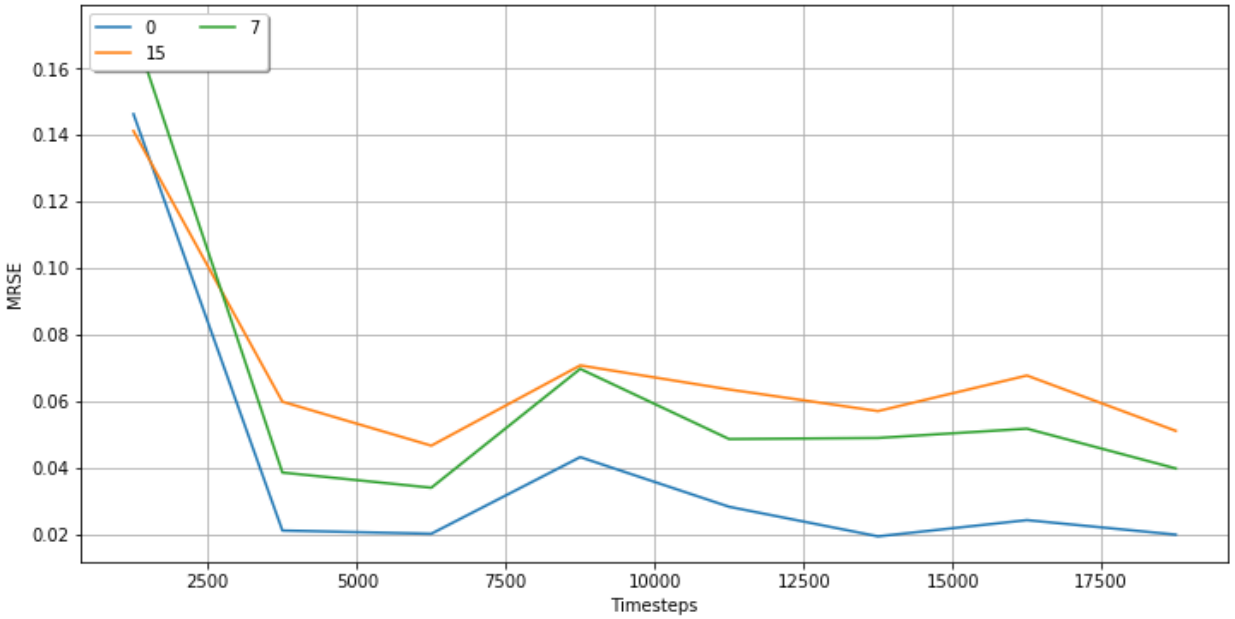

Figure 36: RMSE versus time steps

7.4.3.4.4 Test 0.3

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

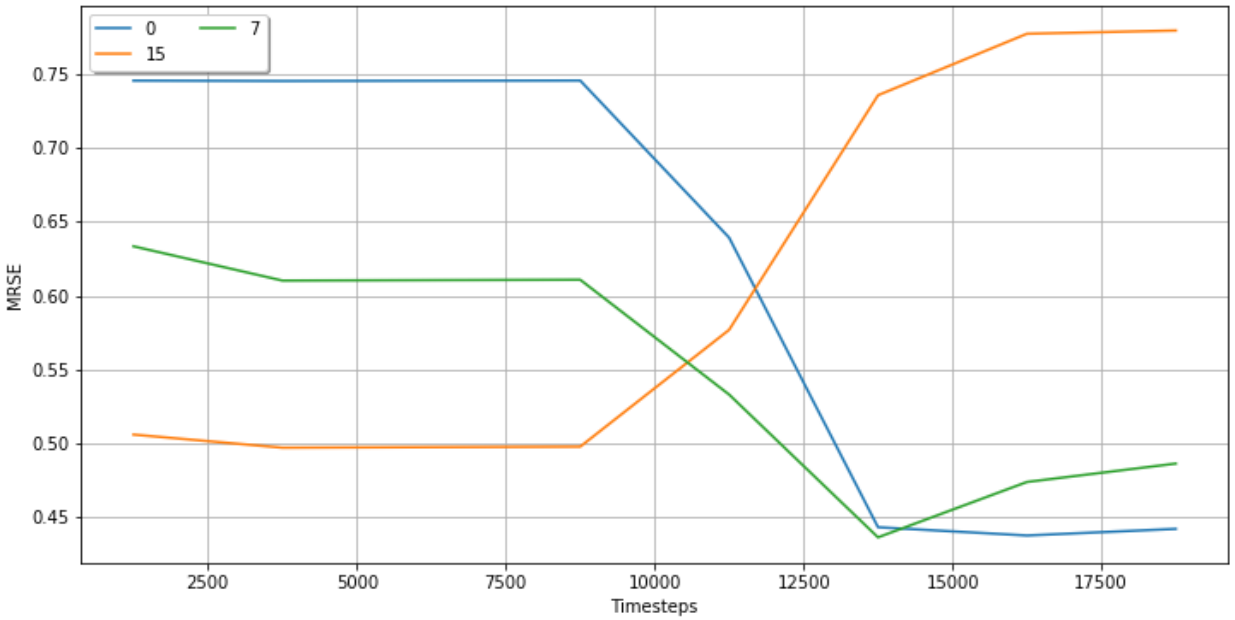

Figure 37: RMSE versus time steps

7.4.3.4.5 Test 0.4

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

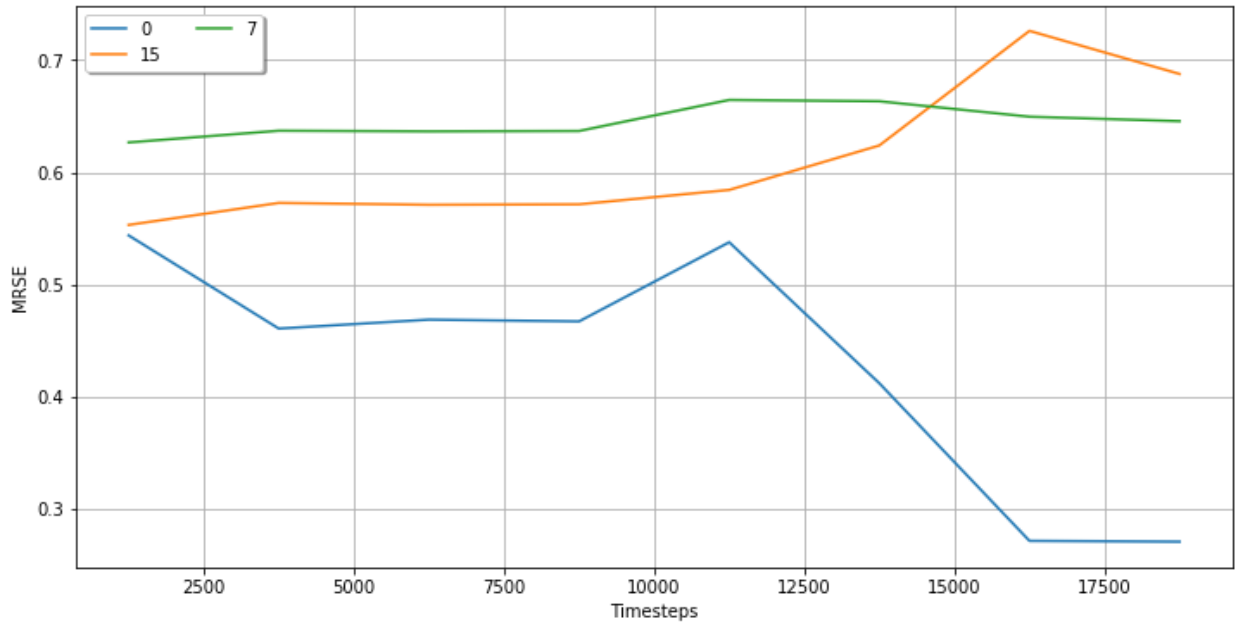

Figure 38: RMSE versus time steps

7.4.3.4.6 Test 0.5

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

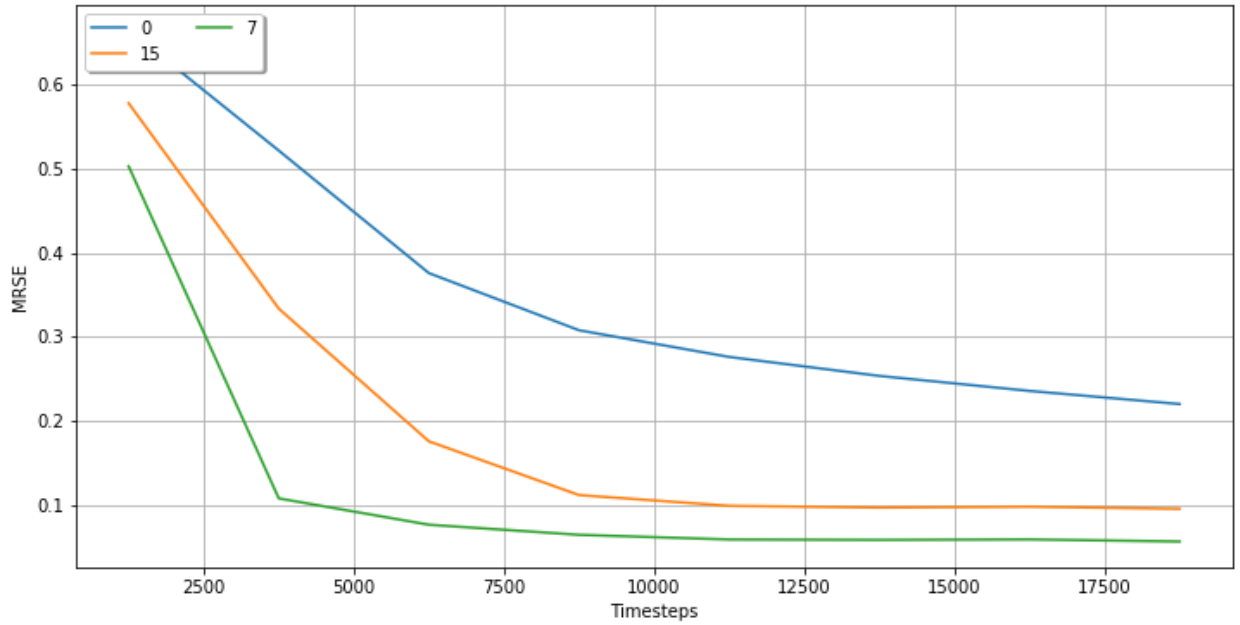

Figure 39: RMSE versus time steps

7.4.3.4.7 Test 0.6

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

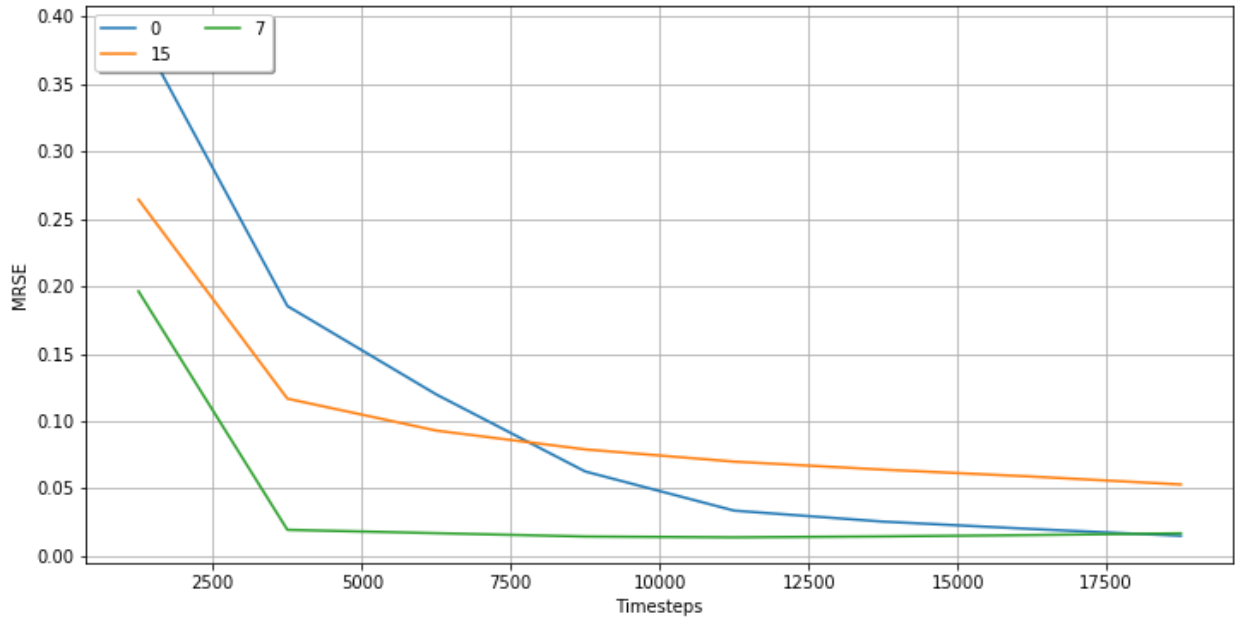

Figure 40: RMSE versus time steps

7.4.3.4.8 Test 0.7

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

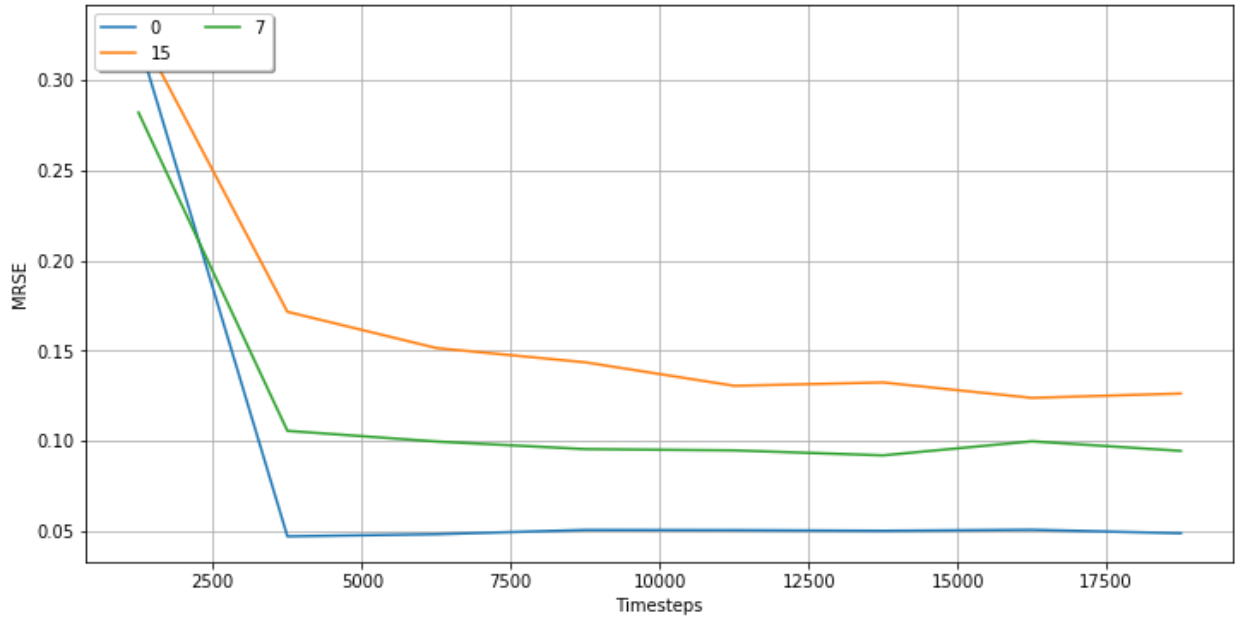

Figure 41: RMSE versus time steps

7.4.3.4.9 Results

The Table 12 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).

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%)

Table 12: test round 0, summary

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.

7.4.3.5 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.

7.4.3.5.1 Test 1.0

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

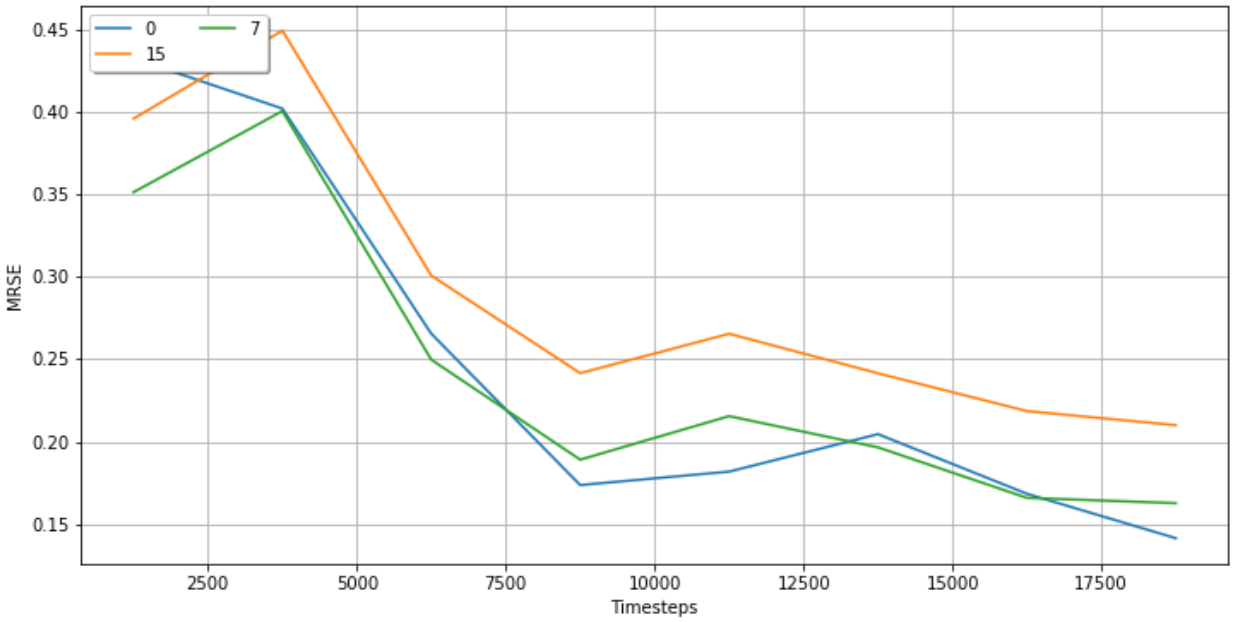

Figure 42: RMSE versus time steps

7.4.3.5.2 Test 1.1

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

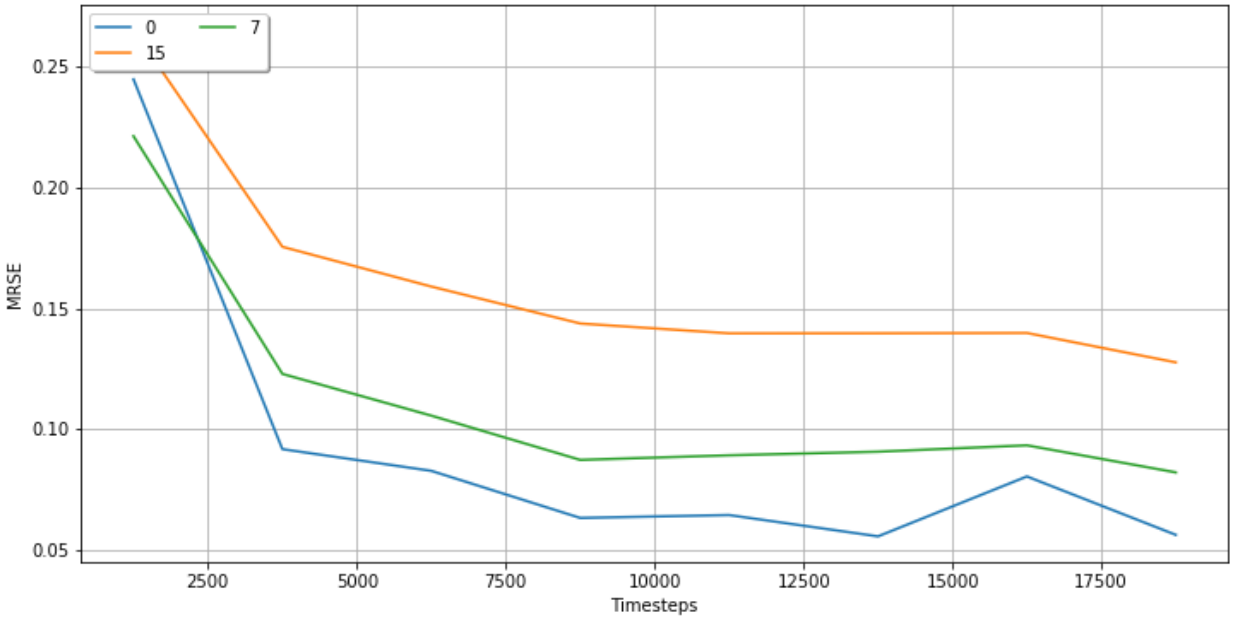

Figure 43: RMSE versus time steps

7.4.3.5.3 Test 1.3

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

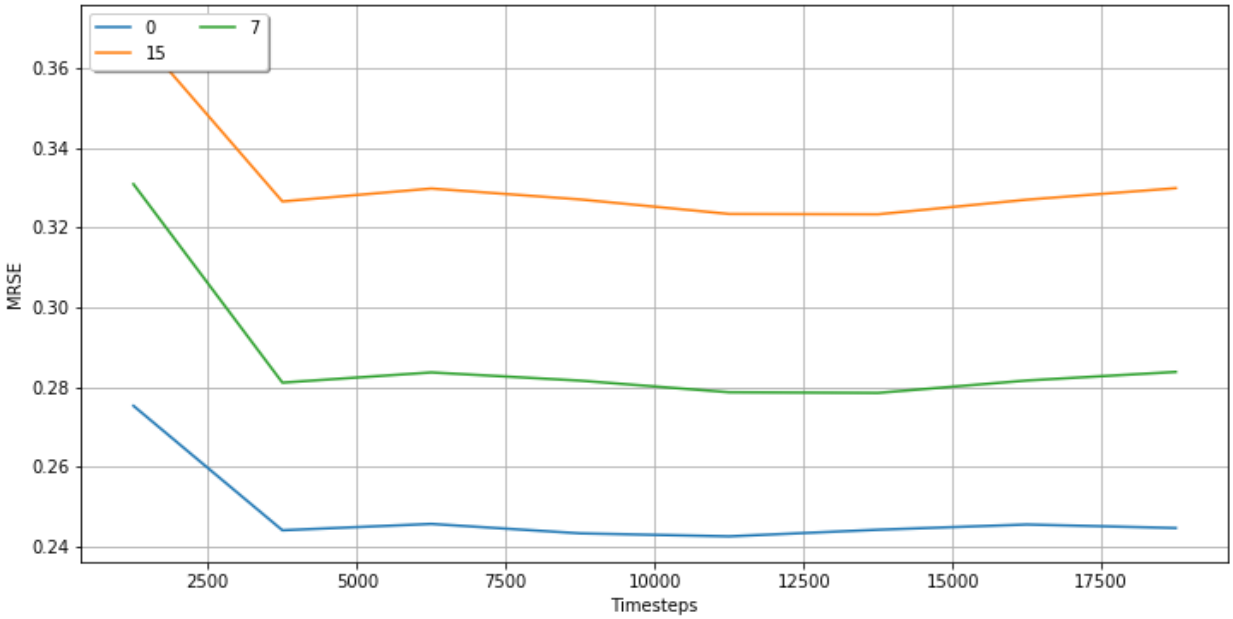

Figure 44: RMSE versus time steps

7.4.3.5.4 Test 1.6

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

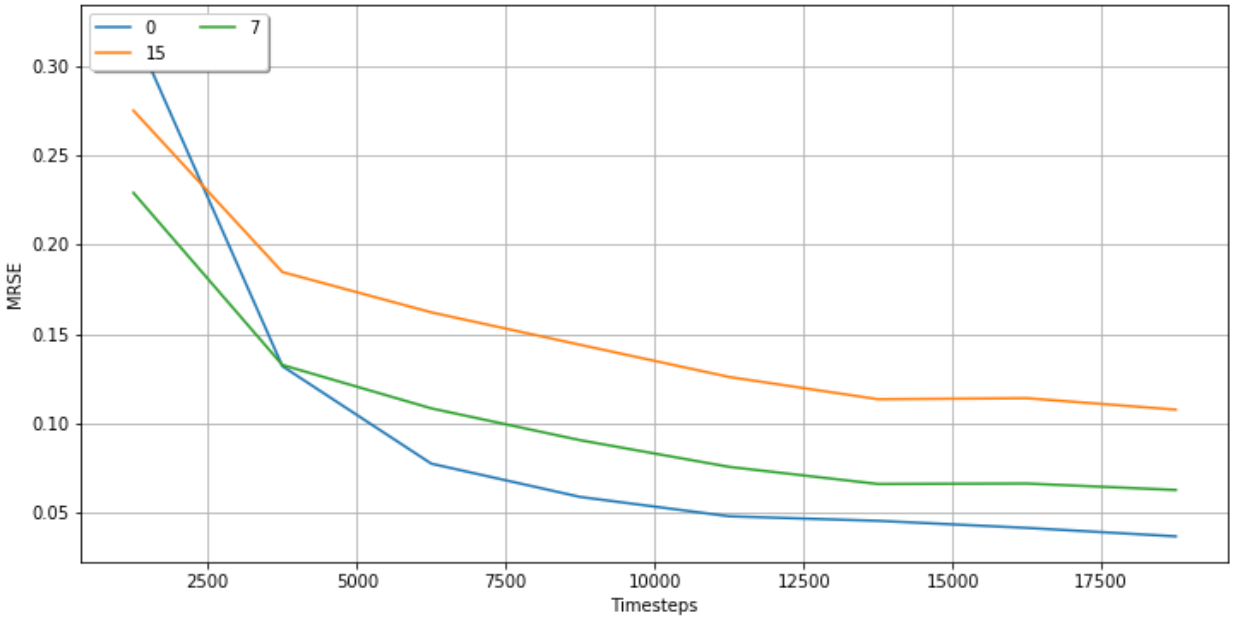

Figure 45: RMSE versus time steps

7.4.3.5.5 Test 1.7

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

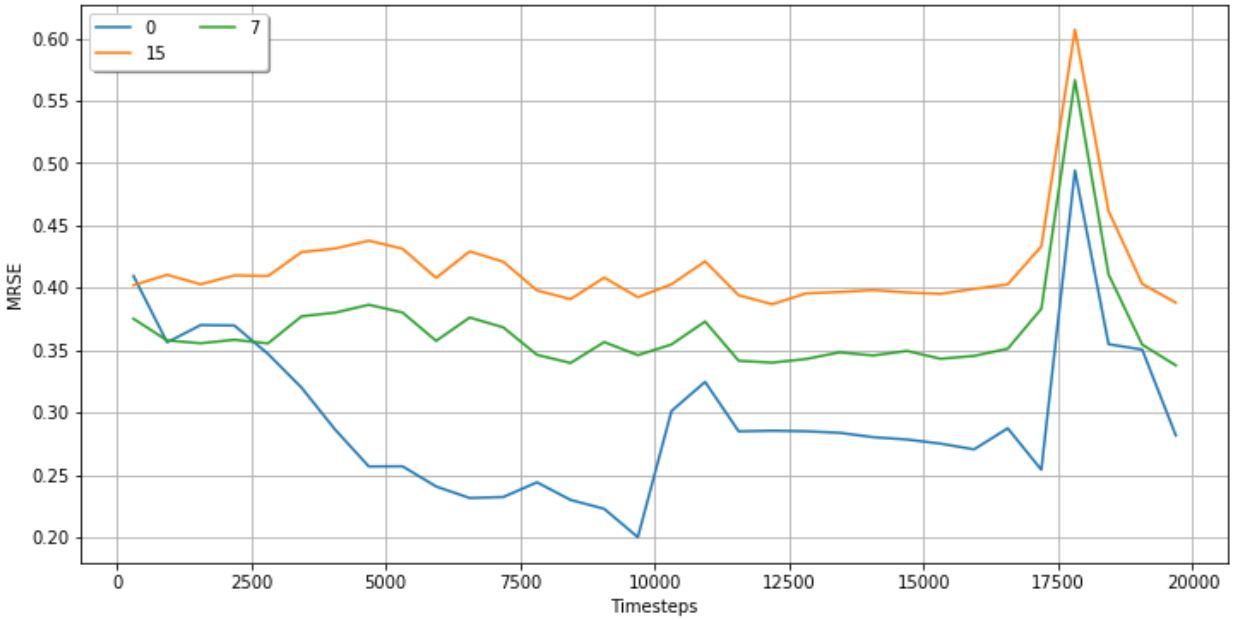

Figure 46: RMSE versus time steps

7.4.3.5.6 Results

The

Table 13 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).

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%)

Table 13: test round 10, summary

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.

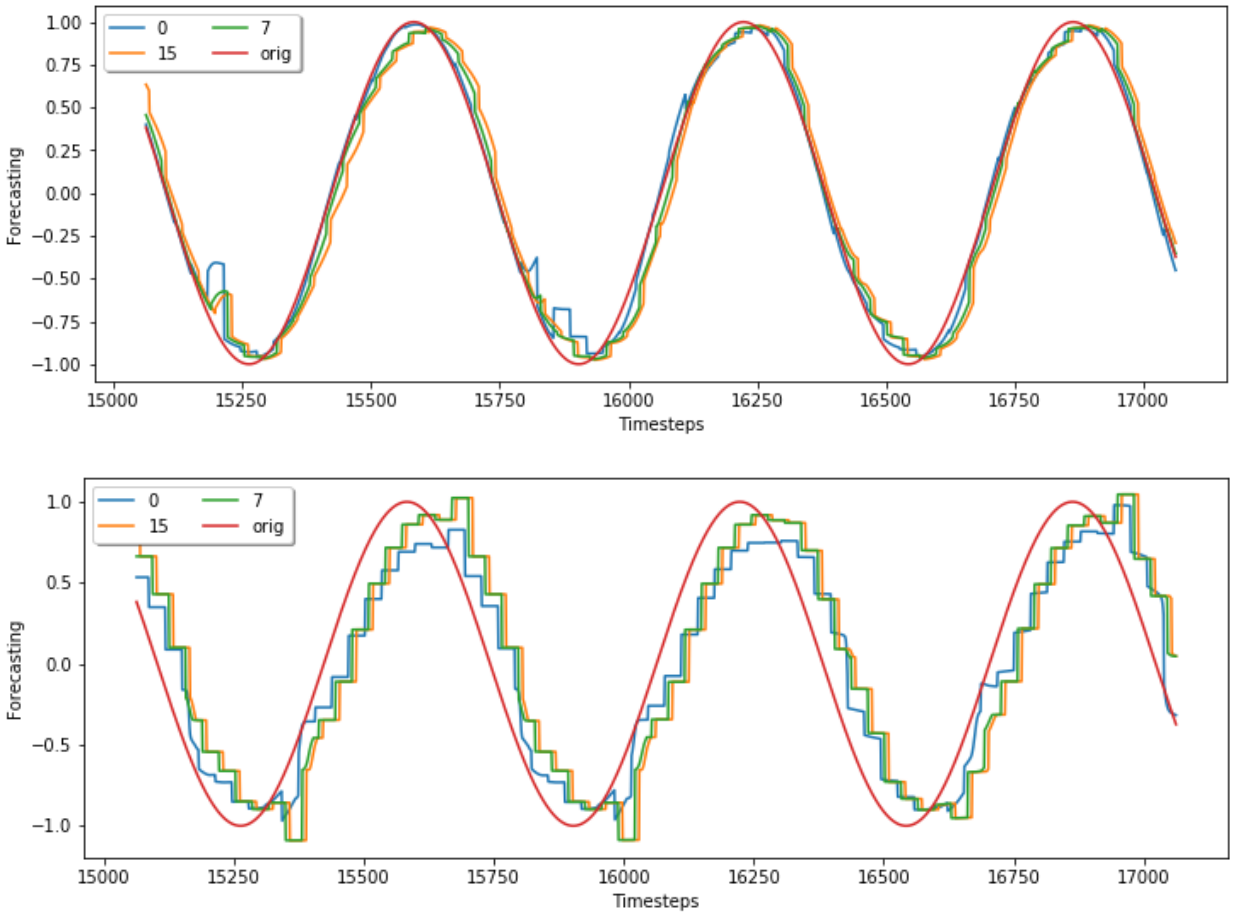

Table 14: target (“orig”) and predictions (“0”, “7”, “15”) at convergence, for tests 1.1 and 1.7.

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.

7.4.3.6 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.

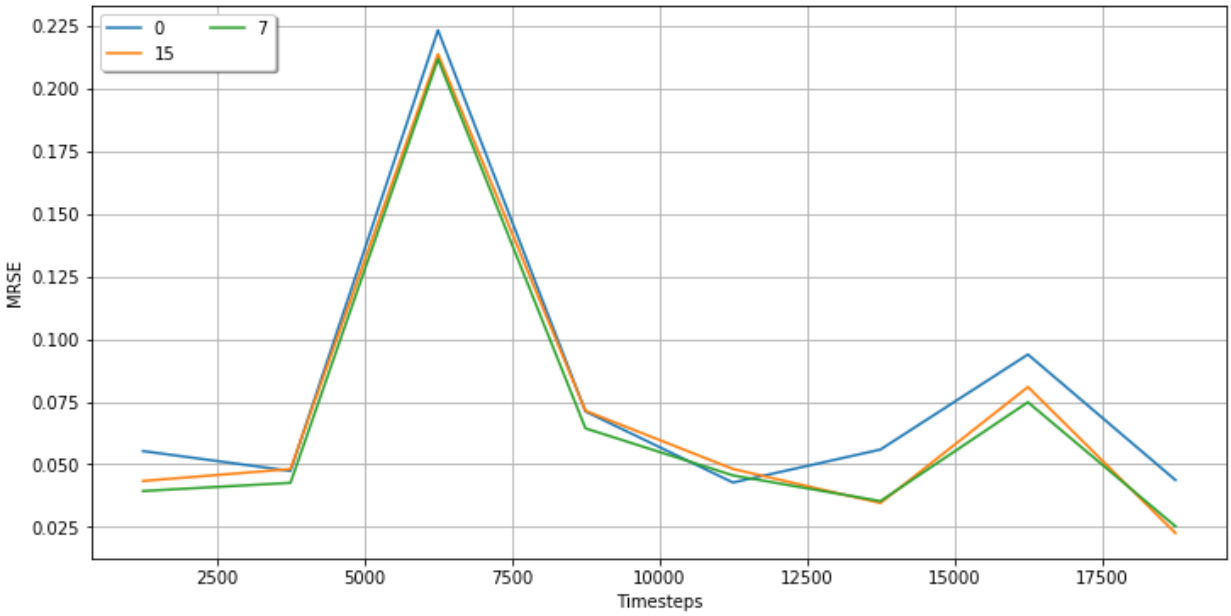

Figure 47: RMSE versus time steps

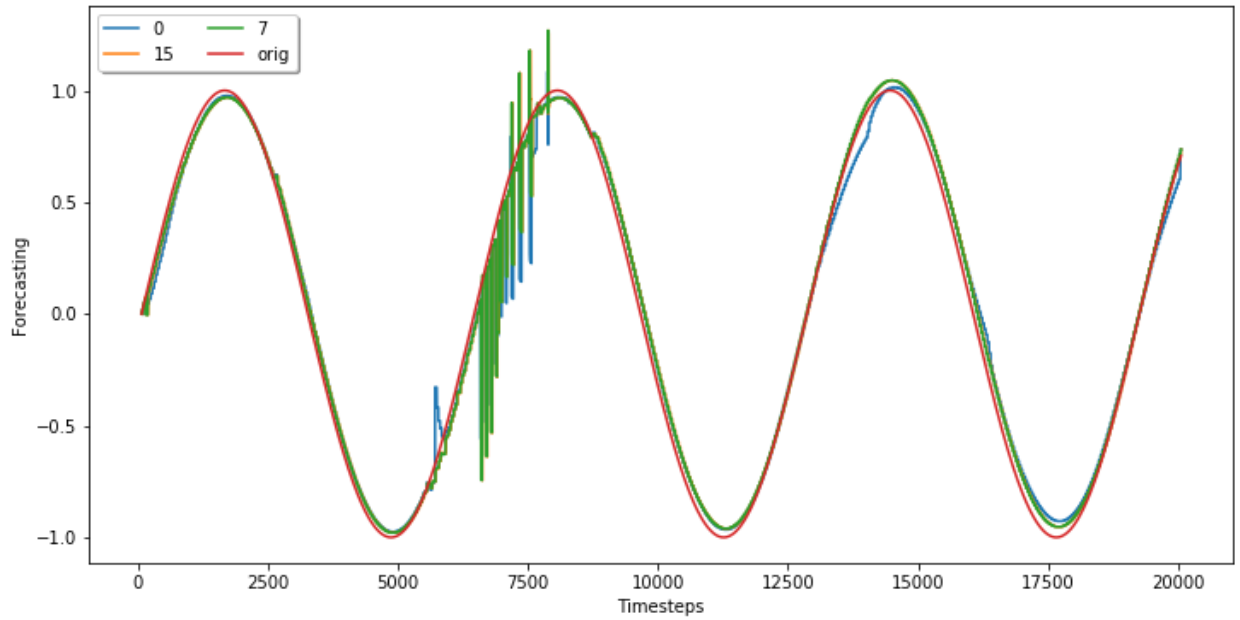

Figure 48: target ("orig") and predictions ("0", "7", "15") for test 2.1.

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.

7.4.3.7 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 49 and Figure 50, convergence was achieved around time step 10000 with a RMSE for the next step prediction of 0.025 (1.25%) which is very good.

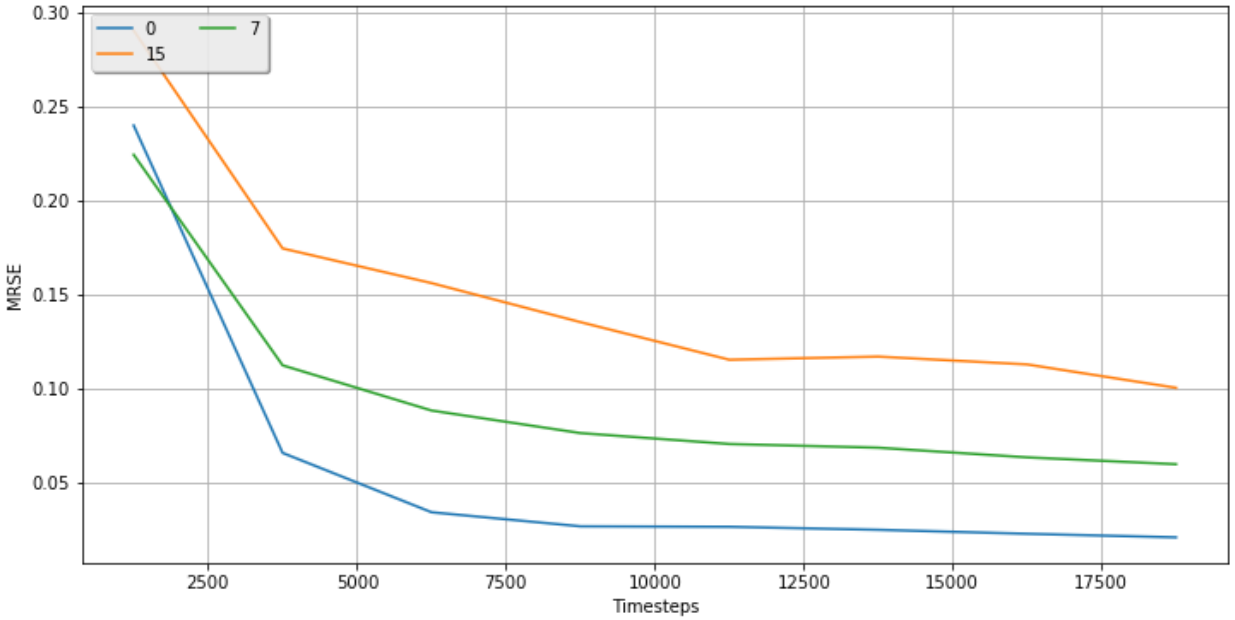

Figure 49: RMSE versus time steps

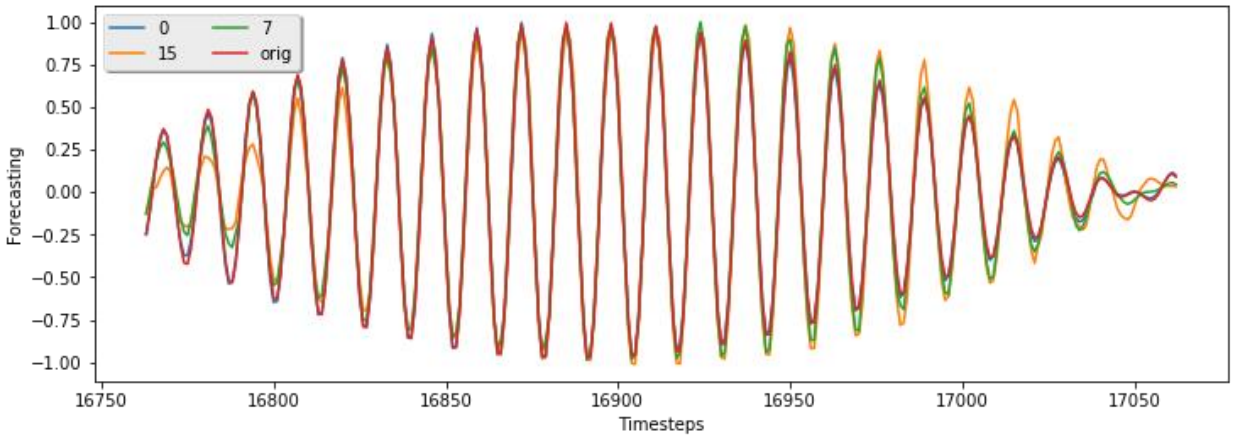

Figure 50: target (“orig”) and predictions (“0”, “7”, “15”) for test 3.1.

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 $0.1T_{max}$.

7.4.3.8 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.

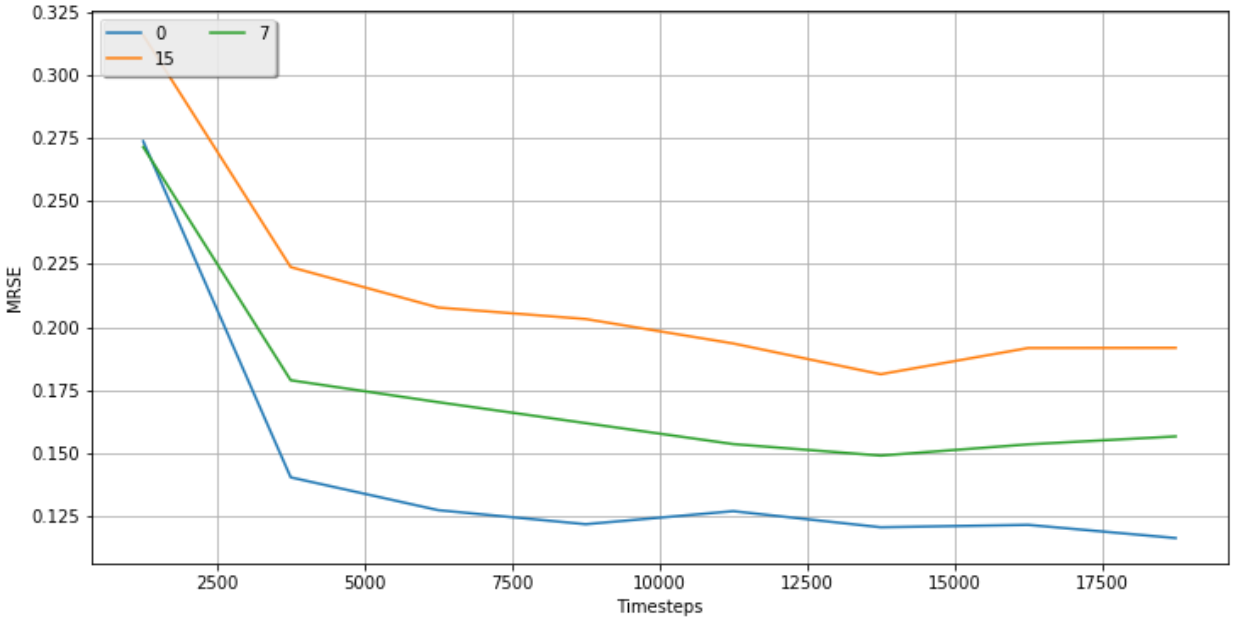

Figure 51: RMSE versus time steps

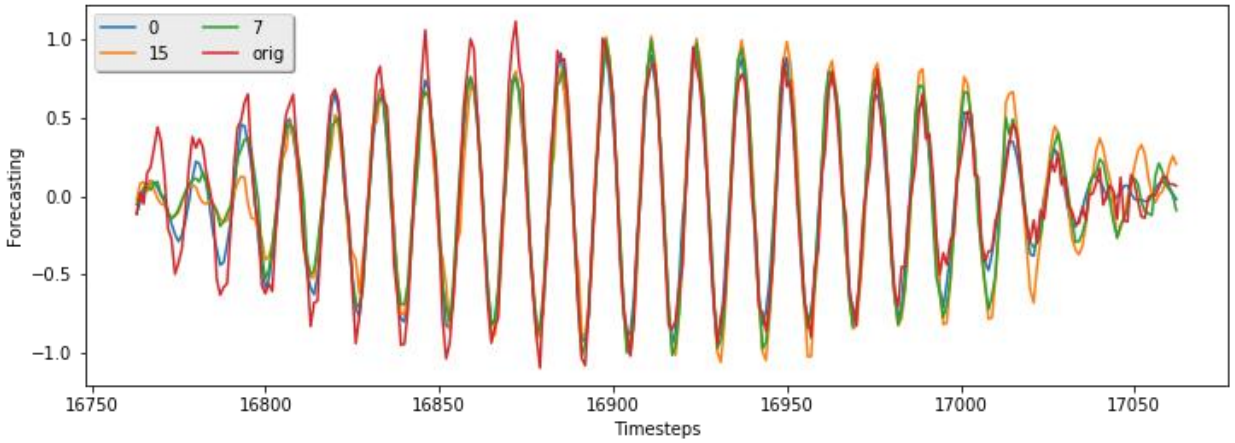

Figure 52: target (“orig”) and predictions (“0”, “7”, “15”) for test 4.1.

The results displayed by Figure 51 and Figure 52 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.

7.4.3.9 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 [12] 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).

8 Conclusions and future work

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 that has taken place have been divided in two typologies. The first kind of activity 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 scraps metal price prediction. The second activity 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.

It is possible to include as main task achievements, the identifications of LSTM as the main sources of models for fitting the required scenarios. Also, Tensor Flow have been chosen as the framework to be used, 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, whilst inputted with a periodic sinusoidal signal, in their untrained version, confirming the choice made.

Future work foresees the analysis of data coming from the newly deployed sensors by designated partners in project's pilot sites. This is mainly due to the considerations made in the data assessment (chapter 6), in which available historical data have been declared as unfit to be used as-is for forming meaningful datasets. Despite the deployed models and topologies, the contribution of data coming from these newly deployed sensors could lead to a breakthrough, accuracy wise. Of course, as previously stated, there are two main obstacles to embracing new sensors into the game:

- The balance of classes is compulsory at any given time when recording historical data. Therefore, the amount of machinery failures required, in a task such as the predictive maintenance, has a cardinality that on average, is higher than COMPOSITION project duration.
- In any activities that performs a continuous learning task, datasets need to have consistency between historical data and live data streams. Therefore, there would be no reference in time of the newest sensors data in already collected historical records. Their relevance would then be negligible, compared to the meaningfulness of the newest sensors. Hence, either untrained artificial neural networks, that as demonstrated in section 7.4.3 are able to converge in a finite time span, will be deployed or a novel historical dataset is required to be recorded. In both cases, the first constrain for having balanced classes remains, making convergence time for matching required accuracy, unlikely in COMPOSITION project time.

Finally, in light of the above considerations, concerning any use case that addresses classification problems for which a suitable balanced dataset is not available, an untrained LSTM artificial neural network is going to be deployed with the analytical certainty that predictions will converge to required accuracy over time. On the other hand, use cases related to prices predictions and the inter-factory scenarios will be probably be able to foresee trained artificial neural network deployed, provided that stationarity trends in current datasets case and any kind of fluctuation starts to appear in the trends, otherwise sporadically changing data that remain constant for most of the time, are well known to be the most unpredictable kind of trends.

In conclusion, the results are promising, technology wise, and advancements with respect of the current state-of-the-art have already been made in this first delivery and clearly reported in section 7.4.3. It is also clear that within the consortium the data available collected independently by end users, years ahead the project starting date, are not always sufficient to determine useful historical datasets by themselves. Hence, it is clear that the power of data analytics resides in the data ownership. The aim is to deploy for the second delivery of task 5.2 (reported in the second version of this document, namely Continuous Deep Learning Toolkit for real time adaptation II, due at M30), tailored models of LSTM Artificial Neural Networks to be used mostly untrained, for each use case that have been analysed, addressing therefore both intra and inter-factory scenarios.

9 List of Figures and Tables

9.1 Figures

9.2 Tables

10 References

[1] T. Mitchell, Machine Learning, McGraw-Hill International Editions, 1997.

[2] “Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Artificial_neural_network.

[3] F. Rosenblatt, “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain,” in *Psychological Review, 65, 386-408*, 1958.

[4] R. Frank, Principles of Neurodynamics, Washington: Spartan Books, 1962.

[5] “Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Recurrent_neural_network#Fully_recurrent.

[6] “Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Vanishing_gradient_problem.

[7] “Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Long_short-term_memory.

[8] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” in *Neural Computation. 9 (8): 1735–1780*, 1997.

[9] S. Haykin, Neural Networks and Learning Machines, Hamilton, Ontario, Canada: McMaster University, 2009.

[10] “Gas dataset,” [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+exposed+to+turbulent+gas+mixtures.

[11] S. C. C. a. G. V. S. van der Walt, “The NumPy Array: A Structure for Efficient Numerical Computation,” *Computing in Science & Engineering, vol. 13, no. 2, pp. 22-30,* vol. 13, no. 2, pp. 22-30, 2011.

[12] “London metal dataset,” [Online]. Available: https://www.quandl.com/data/LME-London-Metal-Exchange.

[13] J. W. a. Z.-H. Z. Xu-Ying Liu, “Exploratory Undersampling for Class-Imbalance Learning,” in *IEEE Trans. Syst., Man, Cybern. B, Appl. Rev,*, 2009.

[14] “Scipy reference,” [Online]. Available: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.save.html.