Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Luís Flávio Loureiro dos Santos

Use of Federated Learning and Neural Networks for Equipment Failure Detection

Master's thesis in Reliability, Availability, Maintainability and Safety RAMS
Supervisor: Shen Yin
June 2023

NTNU
Norwegian University of Science and Technology

Luís Flávio Loureiro dos Santos

Use of Federated Learning and Neural Networks for Equipment Failure Detection

Master's thesis in Reliability, Availability, Maintainability and Safety
RAMS
Supervisor: Shen Yin
June 2023

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of
Science and Technology

Preface

This thesis presents an investigation into the association of Federated learning and artificial neural networks, including a literature about their origins and their potential application in prediction of remaining useful life of industrial equipment. These studies were conducted during the spring semester of 2023 as a required component of the RAMS[1] Master's Programme at NTNU.

The motivation for this study is the need for early detection of equipment failures using machine learning techniques distributed across plants. The author aimed to achieve this while preserving data privacy, and minimizing network traffic by avoiding the transfer of raw data to a central server. The machine learn parameters could be updated whenever new data becomes available.

An example application of Federated Learning is to train models across different vessels, such as oil production platforms or LPG, LNG carriers, located in various regions around the world, with limited or intermittent network access. By training models locally on each vessel and sharing only the training parameters, the amount of network traffic required is significantly reduced. It is also be possible for different companies to benefit from the training using the equipment fleet from all the companies, without sharing raw data, keeping privacy. The authors do not anticipate any significant remote processing challenges, given the computing power available today.

To follow this project, basic knowledge of machine learning and understanding of RAMS concepts are required. Additionally, to implement a federated learning in real-world scenarios, programming skills and familiarity with data manipulation are necessary.

Trondheim, June 2023

Luís Flávio Loureiro dos Santos

[1] RAMS: Reliability, Availability, Maintainability and Safety

Acknowledgment

I would like to thank my wife, Luciana, and my children, Pedro and Daniel for embarking with me in this “adventure”, moving to Norway. I believe it was a valuable experience for our family.

I would also like to thank all the NTNU RAMS Professors for all the shared knowledge. Specially to Professor Shen Yin for the advices, recommendations and guidance through the development of this study.

Also thanks to my parents, brother, relatives and friends for believing and encouraging.

LFLS

Abstract

This thesis aims to study and test the federated learning approach to predict the remaining useful life of operating machines. Federated learning is a method associated with a machine learning process to distribute the training across diferent devices and over time. The first application of federated learning was the “next word prediction” feature of mobile phones’ keyboards. This method was used to train with user’s typing data at the local device, but without sharing the original information, keeping them private. All users benefit from the prediction, without sharing any personal text.

Machine learning processes like artificial neural networks need significant historical data. Remaining useful life prediction usually demands sensors’ data over time and information when failures occur. Once the method is adequately trained, the system should be able to estimate when the failure mode is developing and when the failure is likely to happen.

Federated Learning stands out from traditional machine learning methods by enabling the training of machine learning models on multiple clients while keeping the original data private. The results are then aggregated on a central server, without data sharing. One motivation for companies to ensure data privacy is the General Data Protection Regulation (`https://ec.europa.eu/info/law/law-topic/data-protection_en`), while limited or intermittent network access of certain assets, such as ships traveling around the globe, further supports the case for this approach.

The data available for model training is crucial for achieving accurate predictions. For instance, in the case of predicting the remaining useful life of machinery, a group of companies owning similar equipment could utilize data from each other without sharing the original data. All companies could benefit from more precise predictions by sharing only the training parameters

To test the performance of the proposed Federated Learning approach, publicly available data from turbines will be utilized. The goal is to evaluate the model’s accuracy and how different training configurations can affect the predictions. The method could then be expanded to train on real data across different operators of similar equipment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Federated learning is a new machine learning paradigm that empowers multiple devices to collaborate and learn from data while safeguarding data privacy. The use of artificial neural networks is becoming increasingly prevalent in machine learning because of their ability to learn from data by mimicking the functioning and structure of an organic brain.

Advancements in technology have facilitated a significant increase in satellite bandwidth and range. However, with this comes an increase in network usage. Previously impossible tasks, such as video conferences, are now commonplace, even on traveling vessels. For example, an oil production platform or LNG vessel may possess thousands of sensors spread across multiple industrial processes. However, a centralized machine learning prediction method may need to be more practical due to network coverage and bandwidth limitations in remote areas. This is where federated learning comes in. It enables the learning process in the same device that holds the data, with only the resultant parameters being shared.

Machine learning refers to a broad range of techniques to train computers to predict variables. With a plethora of applications, from healthcare (Yue et al., 2018) to leisure (Missura and Gaertner, 2009), the opportunities are endless. However, privacy issues often constrain the data available for machine learning. This is particularly true for healthcare data, and the General Data Protection Regulation (GDPR: `https://commission.europa.eu/law/law-topic/data-protection_en`) and similar regulations in other countries, empower individuals to limit the use and transmission of their data.

Federated learning offers a novel approach to training models using isolated data sets. For instance, a company with similar equipment may need help to develop its own training models. However, using federated learning, group companies can benefit from training with a much larger equipment fleet while only sharing training parameters, without disclosing raw data.

1.1 Background

Reliability studies enhance industrial production efficiency and safety. Predictive maintenance and Remaining Useful Life predictions are becoming increasingly common in all industries, but they rely on data from field sensors. Due to network coverage and bandwidth limitations, privacy and data transfer restrictions can be limiting, particularly in remote locations.

Federated learning has been implemented in predicting the next word on smartphone keyboards, and can be repeated periodically for improved accuracy. Guo et al. (2022) present a cloud collaboration model for predicting remaining useful life, where multiple clients extract and send low-level features from their own data to a cloud server managing the model. Training is completed on the server side to reduce the client workload, and the method was compared with other methods using the same dataset.

1.1.1 Related work

Several different application are already researched, from user experience enhancements in mobile phones to industrial equipment cases.

Secure aggregation protocols (Hard et al. (2018)) can protect privacy in federated learning, enabling machine learning without sending data to a centralized processing system. Decentralized training can be done directly on the devices with the data, including small gadgets such as smartphones, and the resulting data can be exchanged without privacy issues.

Probably the most known application of federated learning is to predict the next word on smartphone keyboards, as described by (Hard et al. (2018)). Training is done locally on the phones to maintain user's privacy and overcome network transfer constraints. A server aggregates parameters and the final result is shared with all the devices using the application. This process can be repeated periodically to improve prediction accuracy over time.

Bonawitz et al. (2017) implemented an application-generic approach with secure multiparty computation (MPC). This work focuses on implementing privacy-preserving machine learning, even with active adversaries attempting to access the data.

Few RUL prediction literature sources use federated learning, but the ones available are quite recent. This means that this is an application still under development and research, and we can expect to have new developments in the near future.

Guo et al. (2022) address client's potentially weak computing capacity of clients for remaining useful life prediction using federated learning. The clients participate in the training, sharing the low-level features and parameters with the server. The server, a more powerful computer, then completes the training aggregating information from all the clients.

During the creation of this thesis, Du et al. (2023) published about using "Trans-lighter" architecture, which is a light-weight federated learning method for remaining useful life predic-

tion, with the advantage of outperforming other RUL prediction methods.

1.2 Objectives

The aims of this thesis can be outlined as follows:

1. Conduct an exploration of the available literature concerning federated learning techniques.
2. Research existing federated learning methods applied to reliability studies and remaining useful life estimation.
3. Develop a simple script utilizing a federated learning method to estimate the remaining useful life of equipment by dividing the training across multiple instances and simulating their aggregation over time.
4. Determine which parameters can potentially enhance the outcomes of the script.
5. Gather data on equipment failures and identify the optimal parameters to be employed for the training.
6. Evaluate the effectiveness of the script using equipment data.
7. Analyze and explore the obtained results.
8. Recommend possible avenues for future research that can be explored.

The scope of this work is to check the efficacy of a specific implementation of federated learning in conjunction with an artificial neural network by subjecting it to an assessment in a controlled scenario.

It is important to note that the primary focus of this study is not to implement in real-world scenarios. Nonetheless, a more compelling avenue for future research could be to investigate the performance of this model on actual data and implement a case that offers online feedback to the equipment operator.

1.3 Approach

The following methodology is proposed to evaluate the effectiveness of Federated Learning in estimating the remaining useful life. Firstly, gather the necessary data and execute a script using a simple federated learning process with an artificial neural network to determine its capability to predict the remaining useful life of the equipment. Then, conduct a Monte Carlo simulation

by iterating multiple times and changing the order of the equipment used in the training process. This step aims to evaluate the consistency of the results and the expected accuracy using fixed parameters. Only the order of the machines used in the training is randomized during this step. The measurement of this step involves determining the average difference between the predicted remaining useful life and the actual measurement.

Next, repeat the above step multiple times with neural network weights and thresholds. This step aims to simulate the parameters' effects on the network and assess how the training and RUL estimation would behave during the operation. The outcome of this step is a comparison of the predicted RUL values using different parameters.

The dataset used for this study is the CMAPSS Jet Engine Simulated Data (`https://data.nasa.gov/Aerospace/CMAPSS-Jet-Engine-Simulated-Data/ff5v-kuh6/`), which contains multiple simulated datasets featuring various sensors and operational settings that affect the engine's performance.

1.4 Contributions

This work focuses on applying federated learning and artificial neural network to predict remaining useful life. The approach is different from the usual, where the machine learning technique attempts to predict the remaining time before failure. The proposal is to define health states as labels *i.e.*, "Healthy", "Alert" and "Danger", and the objective of the machine learning is then to obtain the most likely state according to the given data. Although each particular subject is already explored, this combination was not found on the researched literature.

1.5 Limitations

The work implements a simulation with synthetic data. It can address a real case problem with some adaptation and access to machinery data. The main limitations of this work are related to the data. Since no operational data was available to the author during the studies, the data used for the training is synthetic data freely distributed at the NASA website (`https://data.nasa.gov/Aerospace/CMAPSS-Jet-Engine-Simulated-Data/ff5v-kuh6/`).

The simulation does not aim to have a ready solution to any remaining useful life problem, but hopefully it may provide guidelines for implementing it. For example, the ideal training parameters and methods for machine learning and data filtering may be different for other applications.

Another limitation of the current work is that the simulation was run on different instances on only one computer. Nevertheless, it is not expected to have different results running in different devices, apart from the processing speed of multiple machines running in parallel. As the

volume of shared data between the instances is small (especially when compared to the data used in training), a network should be a manageable bottleneck. Also, some transmission delays would be fine.

Although data privacy is one of the functions of federated learning, this needs to be explored in depth during the experiments. Original data is particular for all the clients during runtime, but no encryption or other techniques was applied in this context.

1.6 Outline

The structure of this document is detailed below, showing how it is organized and what to expect in each chapter.

- **1 - Introduction:** Introduction to the addressed problems, training methods and the objectives of the work.
- **2 - Theory:** Theory of the methods used to solve the federated learning for the remaining useful live prediction problem, including the machine learning methods, and data selection and pre-processing.
- **3 - Used Dataset:** Shows the details about the used dataset, what it represents, and how to interpret it.
- **4 - Proposed Method:** Presents how the solution was implemented, detailing the main steps, the process flow and the flow results.
- **5 - Discussion and Conclusions:** Shows details on the simulation process, comparison of different parameters, and what can be concluded with the results.
- **6 - Ideas for Further Work:** Suggests next steps for further study and development. It addresses some of the limitations of the current work.
- **Bibliography.**
- **Appendix:** Lists the Python scripts used. They can be freely adapted, improved and expanded for further studies.

Chapter 2

Theory

Machine learning refers to computational methods for making predictions without explicitly programming of the system. This approach is particularly valuable when a model of the system needs to be discovered or is more complex to be constructed. However, it does require training data to enable the system to make predictions. Generally speaking, the more data that is used, the better the system's predictions become.

Perhaps the simpler prediction method to be explained is linear regression. Figure 2.1 shows an example of a linear regression on "original data" and some predictions. If a straight line can model a specific dataset with X and Y pairs, having a new value for X, a corresponding Y value is expected to be in the line. Ideal gases are an example of linear relationship. A regression can be calculated if a couple of pressures and temperatures are obtained on a closed tank containing an ideal gas. With one temperature, it is then possible to estimate the pressure.

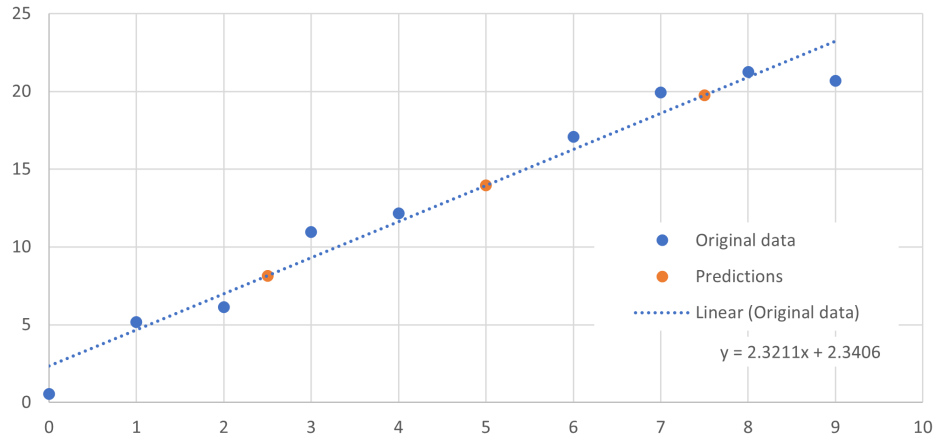


Figure 2.1: Dataset linear regression and predictions

Of course, a linear relationship does not represent all the cases, and different methods were developed for other data types. In this work, the objective is to use one type of machine learning

with federated learning to predict the remaining useful life of turbines based on sensors' data.

Instead of numerical predictions, machine learning techniques can also run classification processes. In this case, information related to the shapes and sizes of flowers may be used to predict what is the plants' species, for example.

2.1 Remaining Useful Life

Rausand et al. (2021) defines the remaining useful life (RUL) as "a random variable that measures that measures the time from t_j until the item is not "useful" anymore". The RUL distribution (equation 2.1) may be obtained if the equipment degradation model is known.

$$Pr\left(RUL\left(t_j\right) \leq t\right) = F_{\mathrm{RUL}(t_j)}(t) \tag{2.1}$$

where: $RUL(t_j)$: remaining useful life for time t_j

When no degradation model is available for the equipment or the failure modes of interest, data-driven approaches may be used. In this document, it is used Data-driven prognostics. Past data is used to predict the remaining useful life with the use of e.g., machine learning. In this case, sensors' data that have some correlation with the failures are needed. Another requirement is that the data needs to contain time-related information up to the failure.

Two approaches may be taken:

- system to predict the estimated useful life for each timestamp
- system to predict a health-related status for each timestamp

This document's approach is to try to predict in which of three states the equipment at each moment, as detailed below:

- Healthy: Ideally, no failure is expected in the near future. The recommendation is to continue monitoring with no special action
- Alert: Equipment has degraded, but it is still safe to operate normally. The recommendation is to take action to prepare for maintenance. The difficulty of equipment acquisition/maintenance, logistics, and the importance of the equipment to safety or lost profits should be taken into account
- Danger: Equipment is highly degraded, and it is expected to fail in the near future. The recommendation is to program maintenance. Again, depending on the importance and production modes, a decision may be taken to carry out maintenance to avoid failure or to have all the resources available immediately after the failure.

This approach offers the benefit of estimating the probability of failure in the upcoming days following the detection of a hazardous state. This aspect will be explored in the next section.

Normally, there is a safety risk related to equipment failure. It would be desirable to perform the maintenance as soon as possible because the failure impact may be significant. If the failure has no safety impact nor other costs, but of the unavailability, to be prepared for maintenance and run until failure may be a strategy.

2.1.1 Probabilities and Decisions

Any machine learning prediction is associated with uncertainty and it is important to be aware of it to make decisions. Suppose the probabilty distribution function (PDF) of an equipment failure is given by $f_T(u)$. In that case it is possible to obtain its reliability $R(t)$ (or survival probability) and failure probability $F(t)$ at time t, using equations 2.2 and shown in Figure 2.2.

$$\begin{cases} R(t) = P(T > t) \\ F(t) = P(T \leq t) \end{cases} \tag{2.2}$$

The probability of failure before time t is also the area under the PDF (or the cumulative distribution function - CDF), of the distribution, and $R(t) = 1 - F(t)$.

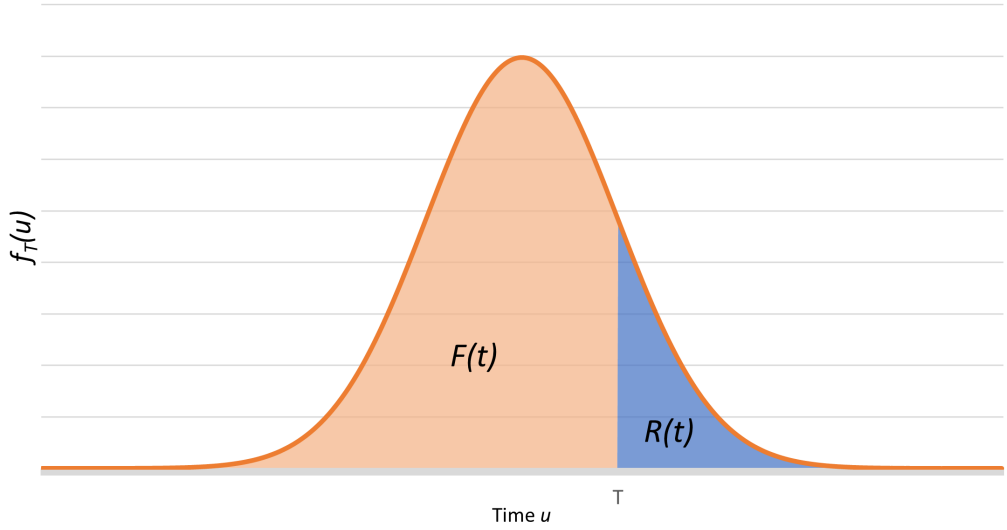


Figure 2.2: Failure Probability and Reliability at time t

If the failure probability function is unknown, a good approach is to build it based on the histogram of failure observations. Figure 2.3 shows one theoretical example. The columns represent the histogram (percentage of failures per time t) of the observed failures, and the line is the estimated cumulative distribution. The failure probability at time t is given by $F(t)$.

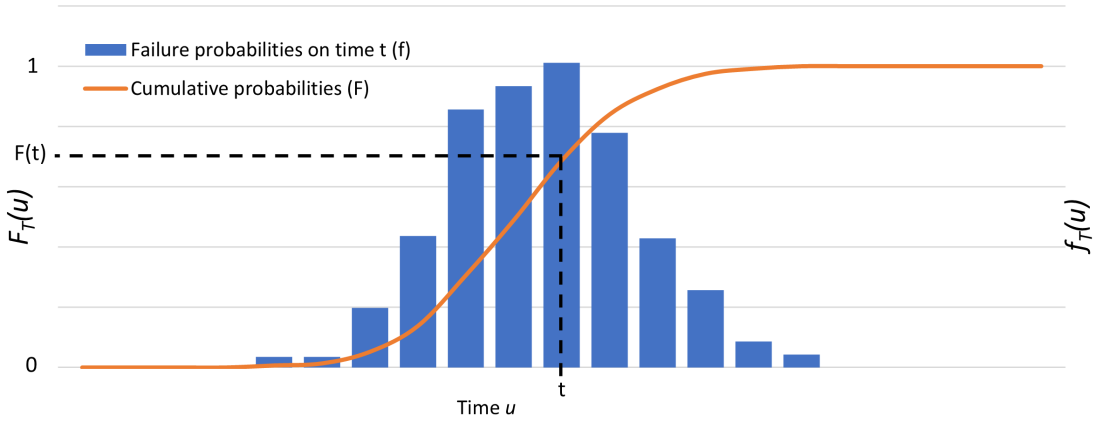


Figure 2.3: Failure Probability and Reliability for observed failure times

With this information, decisions can be made by comparing the failure probability with the accepted chance of failure. The optimization of total costs may define this accepted chance.

2.1.2 Costs

The classic approach to calculating the total life cycle estimated cost, based on a preventive strategy, can be modeled by equation 2.3 and is a function of the failure probability and the costs related to the failure and the predictive repair.

$$C(\tau) = \frac{PM_{\text{Cost}}}{\tau} + CM_{\text{Cost}} \cdot \lambda(\tau) \tag{2.3}$$

$$\text{where: } \begin{cases} \tau\text{: preventive maintenance interval} \\ \lambda\text{: failure rate} \\ PM_{\text{Cost}}\text{: preventive maintenance costs} \\ CM_{\text{Cost}}\text{: cost of equipment repair} \end{cases}$$

The preventive maintenance cost (PM_{Cost}) accounts basically the following:

- acquisition and transport of spare parts,
- mobilization and time of the maintenance team,
- loss of profit for the period.

The equipment repair cost (CM_{Cost}) adds the following costs to (PM_{Cost}):

- additional loss of profit due to the time to unscheduled repair,
- logistic costs due to urgency,

- safety costs, depending on the function of the equipment and its failure consequences,
- additional repair parts depending on the consequences of the failure

Depending on the specific case, different costs may be involved. Often, doing anything different than waiting to fail does not make sense. Usually, that is the lighting systems case. In this case spare parts are cheap, the maintenance is quick, and normally the failure is of little consequence.

But for other types of equipment, the losses, and consequences may be relevant, therefore, therefore beneficial to avoid failures. If it is somehow possible to predict the failure probabilities in the near future of one piece of equipment, it would be possible to replace τ and $\lambda(\tau)$ by probabilities function of time before failure, or after prediction. The probability of not failing could replace τ, while the probability of failure could replace $\lambda(\tau)$. The costs CM_{Cost} can remain unaltered but to reduce the failure probability, PM_{Cost} could increase specially due to logistics time pressure. The ideal time to perform the predictive maintenance after a failure prediction or the probabilities may be calculated by optimizing equation 2.4.

$$C(t_{\text{prd}}) = (PM_{\text{Cost}} + LC(t_{\text{prd}})) \cdot (1 - P(t_{\text{prd}})) + CM_{\text{Cost}} \cdot P(t_{\text{prd}}) \tag{2.4}$$

$$\text{where: } \begin{cases} t_{\text{prd}}\text{: time to planned maintenance after prediction of failure} \\ LC(t_{\text{prd}})\text{: logistics cost, function of } t_{\text{prd}} \\ P(t_{\text{prd}})\text{: probability of failure, function of } t_{\text{prd}} \end{cases}$$

2.2 Artificial Neural Networks

The artificial neural network is a machine learning method inspired by natural neural networks found in animals. In natural systems, Lettvin et al. (1959) explains that these networks propagate signals through neuronal cells. In simpler organic forms as frogs, it allows the animal to directly perceive the environment and react to get food or to escape a predator. In more complex animals, such as superior mammals, including humans, the networks can carry and join information from the different sensorial systems, such as images, sounds, and smells, and information previously learned and memorized to consider alternatives and make decisions through a central system. Information transmission, storage, and processing are based on simple elements (neurons) that can propagate signals to the next elements.

An early attempt to model the natural networks in logical systems was developed by McCulloch and Pitts (1943), where equations could be established to explain simple networks with a few neurons. The neuronal transmissions were modeled as being “all-or-none”, or boolean levels. If the sum of inputs of an element reaches a threshold, this element is activated, and the

signal continues to be transmitted to the ones connected to its output. The objective then was not to model an artificial system but to try to understand and explain the behavior of biological ones. Some years later, Rosenblatt (1958) expanded the model, adding weights to the inputs of each neuronal element, as shown in Figure 2.4, and was able to program an artificial network that could respond to images. Each perceptron (the digital analog of the neuron, as it was called by Roseblatt) can be understood as a simple logic gate that is activated if the sum of the weighted inputs reaches a given threshold. The weights represent the importance of each input activating the elements. Inputs and outputs are modeled as Boolean values.

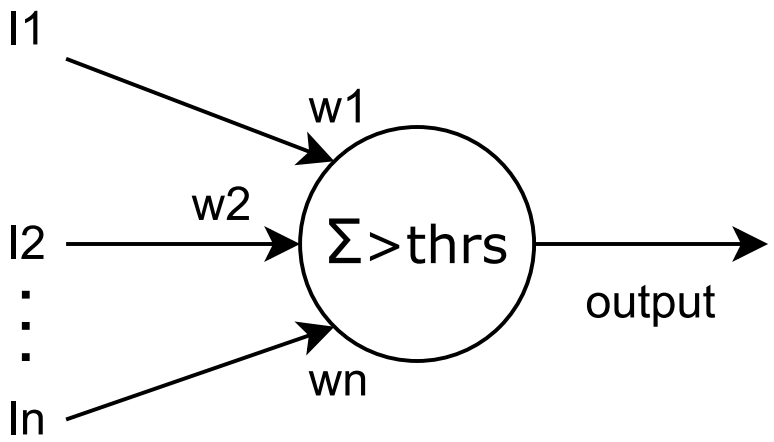


Figure 2.4: Neuron diagram (perceptron). Adapted from Nielsen (2019)

Equation 2.5 shows the basic model for each element.

$$\text{output: } \begin{cases} 0: \text{ if } \sum_{i=1}^{n} I_\text{i} w_\text{i} <= threshold \\ 1: \text{ if } \sum_{i=1}^{n} I_\text{i} w_\text{i} > threshold \end{cases} \tag{2.5}$$

$$\text{where: } \begin{cases} I_\text{i}\text{: elements' inputs, boolean values} \\ w_\text{i}\text{: inputs' weights, real values} \\ threshold\text{: activation threshold for the output of each element} \end{cases}$$

Figure 2.5 represents a network for vision processing. It contains an input layer (projection area) receiving information from the retina, and its connections to an intermediate layer (association area), up to the output layer with the response elements.

Figure 2.6 shows a generic neural network. It may contain any number of layers with any number of elements each. While the decisions made by a single element are very simple, more elements and layers can create extremely complex decision flows.

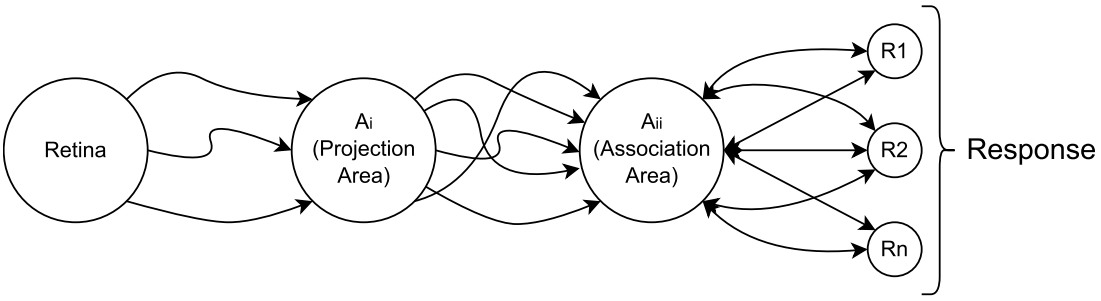


Figure 2.5: Vision Neural Network model. Adapted from Rosenblatt (1958)

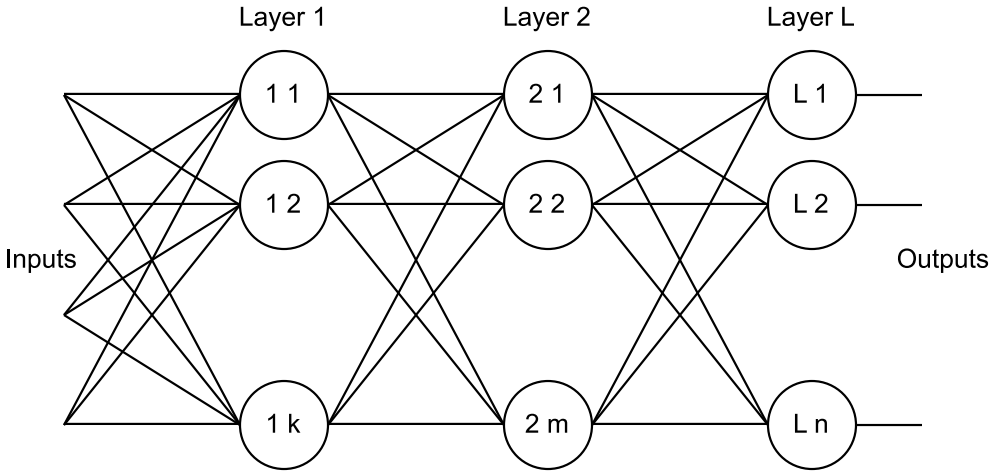


Figure 2.6: Generic Neural Network diagram

With Boolean levels (step function), one small change in weight, for example, may create a big shift in the output of an element and also in the results of a neural network. As Nielsen (2019) explained, activation functions were later introduced instead of boolean levels as outputs to make changes more subtle. One of the most intuitive activation functions is the sigmoid function. It acts as a smooth step function. Figure 2.7 shows a plot comparing the step (equation 2.5) and sigmoid (equation 2.6) functions.

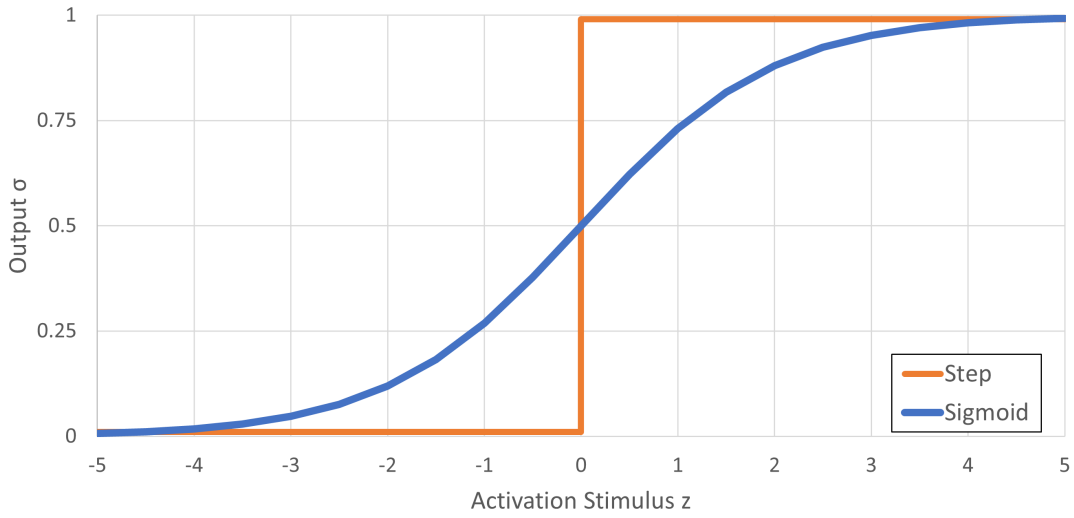


Figure 2.7: Comparison Step and Sigmoid functions

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.6}$$

$$\text{where: } \begin{cases} \sigma\text{: sigma function} \\ z\text{: activation stimulus} \end{cases}$$

Other functions are in use for this project, ReLU (rectified linear unit) and Softmax. The benefits of ReLU are similar to the sigmoid, but its simplicity provides a better performance. Figure 2.8 compares it with the step function and it is modeled by equation 2.7.

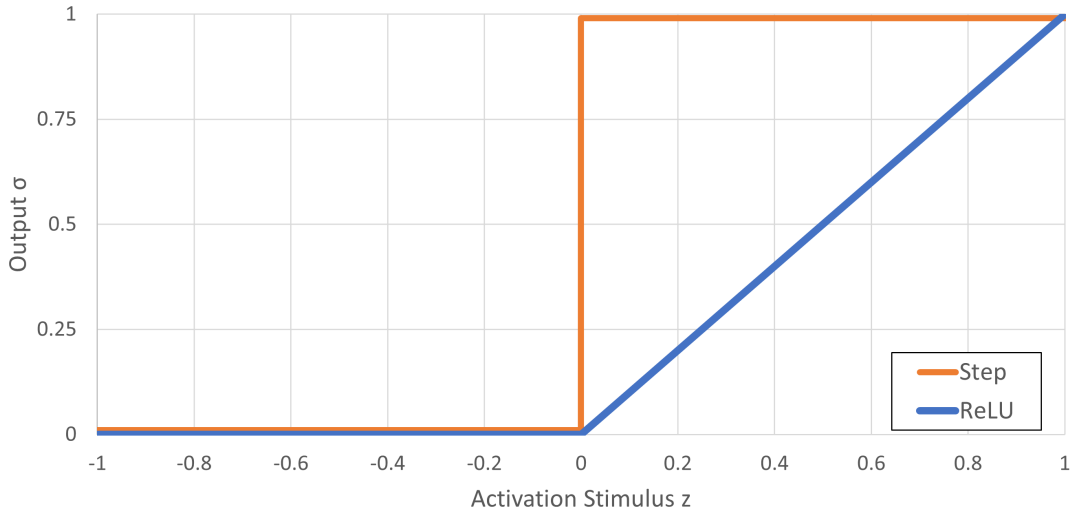


Figure 2.8: Comparison Step and ReLU functions

$$\begin{cases} z \text{ for } z > 0 \\ 0 \text{ for } z \leq 0 \end{cases} \tag{2.7}$$

The softmax function is often used as the last layer in classification problems. It behaves similarly to sigmoid functions, but with it, each neuron responds for a classification label, and it will output a value from $zero$ to one being the probabilities of that label. Its function is given by equation 2.8, and the sum of all the neurons outputs in that layer yields one.

$$softmax(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } j = 1, ..., K \tag{2.8}$$

2.3 Federated Learning

Federated learning trains a machine learning method distributed across several data sources (clients). The objective is that several clients process each training with their own data. Each then shares the training parameters with a server that merges them. Original data is never shared between the training computers and the server. The merged parameters contain training information from all the clients' datasets without sharing the original data. The aggregation of the training parameters from the server is then shared with all the clients. They then benefit from better predictions results based on their own data, and the whole group of clients'.

2.3.1 The Need for Federated Learning

Users' personal data restrictions imposed by laws over the last years added some challenges to all industries. The European General Data Protection Regulation (GDPR - `https://commission.europa.eu/law/law-topic/data-protection_en` states that no personal data may be shared across different entities without their express consent. As a member of the European Economic Area, GDPR is also applicable to Norway. *Datatilsynet* (`https://www.datatilsynet.no/`), the Norwegian Data Protection Authority is a body that supervises any holder of other persons' information to make sure that the legislation is followed. Other countries have similar legislation about data protection.

Several data-breaching scandals go public every couple of months. Social networks, email providers, retail companies, and banks had their leaks at some level. Probably most, if not all, types of industries suffered from this. Even if the consequences of the GDPR laws are not considered, a company's image may be severely affected by such events. Centralizing all the personal data is a risk, as it would be one point of access for a hacker to access and collect all the information. Accessing multiples user's data is much harder if the information is in the users'

possession.

Also, bandwidth limitation, especially in remote devices, such as mobile phones or vessels traveling around the world, impose some restrictions. The sharing of training data may create the need to transfer extremely big files. On the other hand, sharing only the training parameters would need small files. Also, bringing the benefit of improvement of predictions and updating the training parameters over time. Transfering several gigabytes of sensors data from multiple vessels across the globe may take some time. To transfer a few kilobytes of training data from or to them, would probably require less then a minute.

2.3.2 Federated Learning Method

Hard et al. (2018) proposed a way to training user data to avoid data exposure. This was the start of federated learning and is used to train the next-word prediction on mobile phones keyboard apps. Probably most of the users would not want to have their typed texts shared over networks, but they certainly take advantage of the next-word prediction feature. The training is processed using the users' data on their own devices. The training parameters are shared, so it would be impossible to recover the original messages. These parameters from multiple users are then shared with a server that joins them all and shares them with the users periodically, improving the predictions.

Figure 2.9 presents the generic flow of federated learning. The flow includes a cloud server, and three client devices. As the training is carried out across several devices with their own (small) data set, most devices nowadays can process it, from computers to mobile phones.

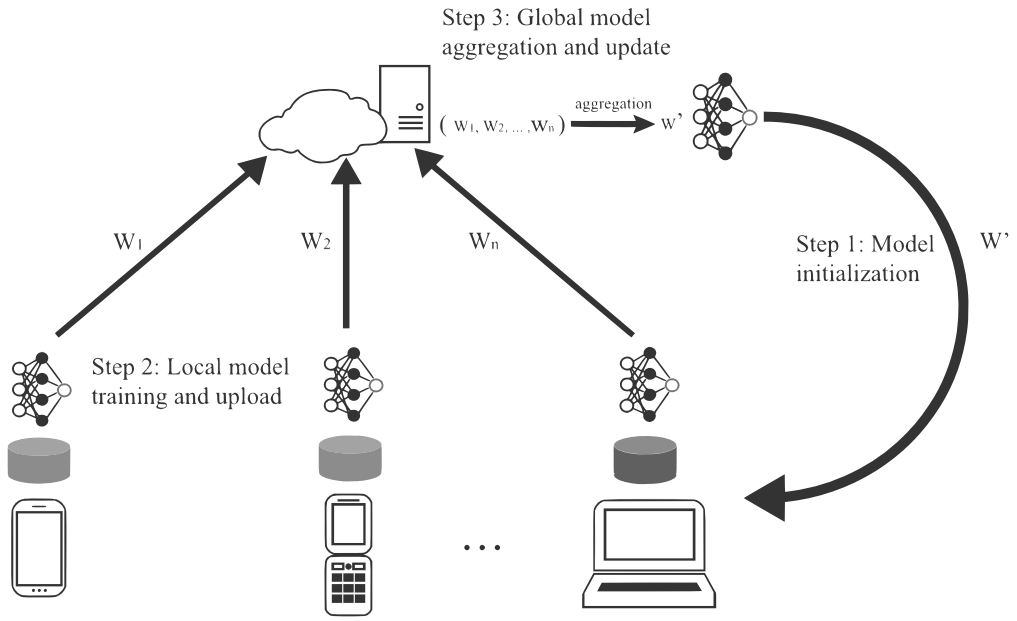


Figure 2.9: Federated learning basic diagram (adapted from Zhang et al. (2021))

2.3.3 Federated Learning Steps

1. The model is initialized with previous training from the (cloud) server and its parameters (W') are shared with all the clients.

2. The clients (any device) uses the last shared parameters on their predictions and train the model with its own data. Each client then has different training results (weights and thresholds in case of a neural network) (W_1, W_2, W_n), and they send them to the server.

3. The server aggregates all the training parameters and shares them back again with the clients restarting the cycle.

2.4 Sensors Signals

Sensor signals are essential for remaining useful life predictions. Having sensors that have poor or no correlation to the failure mode analyzed, as they bring no or nearly no information about the development of the faulty condition. Then a selection of useful sensors is usually needed.

Also, very noisy signals need to be avoided, as they may also disturb more than help in the predictions. This is true because it's impossible to be sure how much of a measurement is due to noise. As sensors' information usually are transmitted analogically with electric signals, they are subject of interference between several sources and other electric transmissions. There are several methods to minimize the noise, but they cannot be completely removed. The use of some digital filters may be needed. Noise usually adds a high frequency, but not sometimes, so each case must be understood to be addressed.

2.4.1 Sensors Selection

One machine may have several different sensors. But it is expected that not all of them will carry useful information about the life of the equipment, and it is usually needed to select the best sensors for the desired prediction. With this objective, there some correlation tests are available that can be used. For machine learning purposes, the sensors with a higher correlation with the desired predictions should be used.

Pearson correlation coefficient

The Pearson coefficient is measures how two series of values are linearly correlated. Its value ranges from -1 and 1. $Zero$ is a total lack of correlation, and the extreme values, perfect correlations. Negative values mean that the correlation is inverse. Equation 2.9 is used to compute this correlation measure.

$$\rho(x,y) = \frac{E[xy]}{\sigma_x \sigma_y} \tag{2.9}$$

$$\text{where: } \begin{cases} E[xy]\text{: cross correlation between } x \text{ and } y \\ \sigma_x^2, \sigma_y^2\text{: variances of the signals} \end{cases}$$

This method is particularly useful if it is known that the expected correlation is to be linear.

Spearman's rank coefficient

Spearman's rank coefficient calculation is essentially the same as the Pearson correlation, using equation 2.9. The difference is that it uses the values ranks of the series instead of directly using the series' values. The result is that it is actually a measurement of how the series ranks are correlated. It is then useful for both linear and non-linear relations.

The values at the equation are then replaced by $E[R(x), R(y)]$, $\sigma_{R(x)}$ and $\sigma_{R(y)}$, where $R(x)$ and $R(y)$ are the ranks of the values of x and y series. The ranks can be understood as the indexes of the ordered series of values.

2.4.2 Signal Filtering

Depending on the measured signal, it may have many different predominant frequencies. Signals related to temperatures usually have a slow behavior, as normally, this variable is subject to heat capacitance. High-speed machines may have very fast vibration measurements, while a discharge pressure of an alternative compressor will usually be between the previous variables.

Figure 2.10 shows a plot with signals of three different frequencies and their composition.

- If the measured signal represents a temperature, a low frequency signal would probably be expected, and high frequency would be related to noise. In this case, the high frequency would need to be filtered out to recover the variable of interest. Using a low pass filter.
- If the measured signal represents a vibration on a high-speed machine, a high-frequency signal would probably be expected, and low frequency would be related to noise. In this case, the low frequency would need to be filtered out to recover the variable of interest. Using a high pass filter.
- If the measured signal represents pressure on a reciprocating compressor, a medium frequency signal would probably be expected, and there could be lower and higher noise related frequencies. In this case, the low and high frequencies would need to be filtered out to recover the variable of interest. Using a band pass filter.

This is a theoretical example, and each case needs to be evaluated. As the noise comes from several sources, different frequencies may be involved needing help to model. But knowing how the variable of interest is expected to behave, it should be possible to filter out the undesired ones. There may be extreme cases where the noise is so important to the composed signal, that the recovery of the measurement may not be feasible.

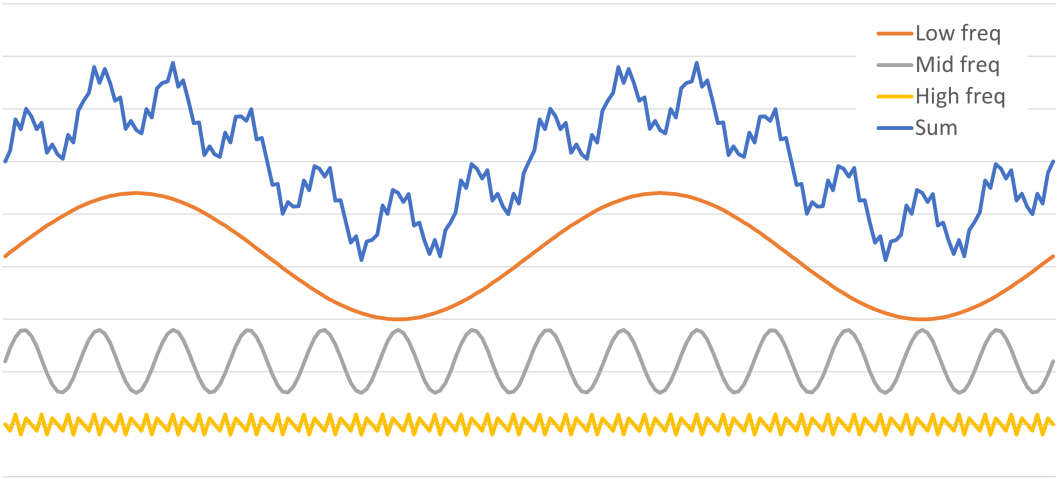


Figure 2.10: Frequency Signals and their composition

Moving average

The moving average is one simple way of smoothing or removing higher frequencies from a signal. The method tends to remove the high frequency by sequentially calculating the average of a number of subsequent data points on the series. Equation 2.10 may be used to calculate the moving average. The equation is generic for the Savitztky-Golay filter (explained in the sequence), where the moving average is a particular case.

$$Y_j^* = \frac{\sum_{i=-m}^{m} C_i Y_{j+i}}{N} \tag{2.10}$$

where:
- Y_i^*: j-th filtered element
- C_i: i-th weight on the rolling window
- Y_i: j-th original element
- N: divisor (denominator)

Consider one sequence of values from a measurement. The filtered value (Y_i^*) at the i-th of the sequence is the average of the value at i-th and its m nearest neighbors. In the moving

average, for equation 2.10, C_i always equals 1 and N always equals the window width $(2m+1)$.

From the composition of signals from figure 2.10, applying the moving average with different widths, it is possible to remove the higher frequency, or recover the lower one signal as shown in figure 2.11, depending on the size of the window.

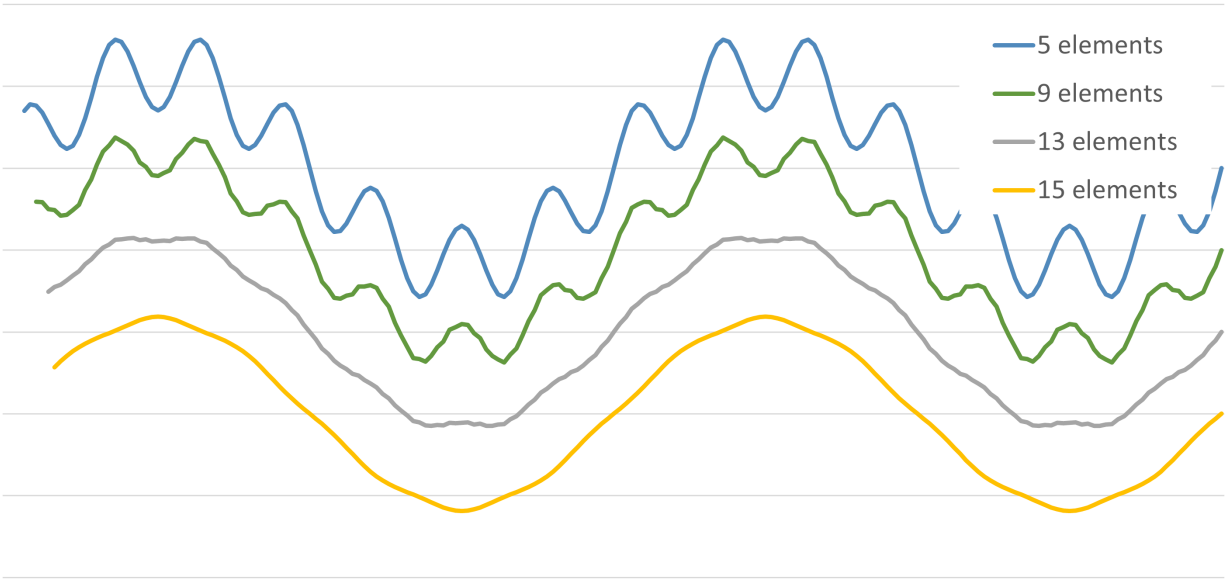


Figure 2.11: Filtering by moving average. 5, 9, 13 and 15 window width

Savitzky-Golay Filter

The Savitzky-Golay filter is a digital filter for a sequence of samples Savitzky and Golay (1964). It uses the same equation 2.10, but in this case, the values for C_i and N are to be changed by different parameters. These parameters fit polynomials and its derivatives to the samples at the moving (convoluting) window. The filtered Y_j^* is then the middle point of the fitting polynomials. Figure 2.12 shows examples of filtering the same composed signal from figure 2.10, filtered by different parameters, recovering specific frequencies. Table 2.1 shows the parameters C_i and N for four different polynomials and cases.

Equation 2.11 shows the example of a Savitzky-Golay filter with a quadratic polynomial fitting the five elements on the moving window (Y_{j-2} to Y_{j+2}).

$$Y_j^* = \frac{-3Y_{j-2} + 12Y_{j-1} + 17Y_j + 12Y_{j+1} + -3Y_{j+2}}{35} \tag{2.11}$$

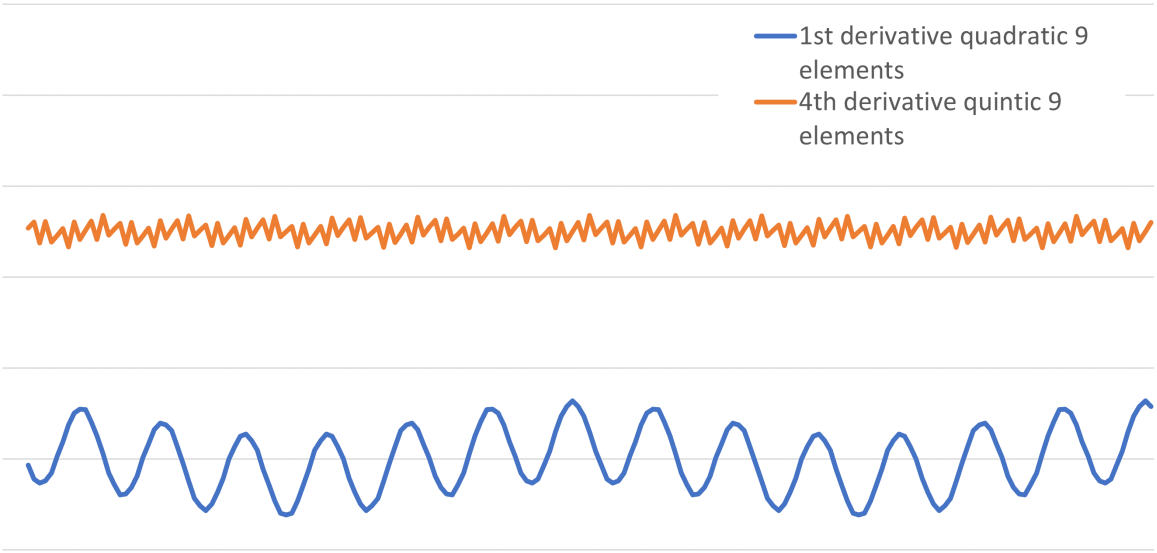


Figure 2.12: Filtering with Savitzky-Golay filter, different parameters

Table 2.1: Parameters for some Savitzky-Golay Filters (Savitzky and Golay (1964))

Polynomial:	**quadratic/cubic**		**cubic/quartic 1st derivative**	
Window size	**5**	**7**	**5**	7
i	Ci	Ci	Ci	Ci
-3		-2		22
-2	-3	3	1	-67
-1	12	6	-8	-58
0	17	7	0	0
1	12	6	8	58
2	-3	3	-1	67
3		-2		-22
Normalisation (N):	35	21	12	252

2.4.3 Signal Normalization

It is usual to have multiple types of sensors measuring several physical variables such as temperature, pressure, speed, vibration, and more. They may assume very different values from each other and have diverse variances that can affect the training and predictions. Usually, a normalization is applied, and probably the simplest one is defined by equation 2.12

$$Sn_{\mathrm{i}} = \frac{S_{\mathrm{i}} - \mu_{\mathrm{train}}}{\sigma_{\mathrm{train}}} \tag{2.12}$$

where:
- i: sample index
- Sn_{i}: normalized sensor sample
- S_{i}: raw sensor data
- μ_{train}: training data average
- σ_{train}: training standard deviation

Chapter 3

Used Dataset

The prediction of remaining useful life needs two types of data. Sensor data over time is normally used, to get information on the degradation of the equipment and also information on when the failure occurred is needed.

In some industries such as offshore Oil and Gas a data collection and storage system is usually implemented during the project. The data is usually collected from the sensors measuring physical variables such as temperatures, pressures and vibrations. As the degradation progresses, these sensors' measurements are expected to change.

Failure information also needs to be collected, which normally needs to be done manually. Although failures may be automatically detected, the failure mode definition is usually discovered during the maintenance procedure. As it depends on people, this is, perhaps, the most difficult information to be reliably obtained. The responsible team needs minimal training and motivation is needed. The development of different failure modes manifests in different ways and sensors types.

To have a useful developing failure detection, it must also detect which failure is being developed, and for this to happen the data for machine learning needs to contain this information on historical data. Different options exist for training a machine learning process to detect different failure modes. It is possible to have one training for each failure mode, and this can be done using different sensors for the cases. It is also possible to train the system to classify what type of failure is currently being developed.

The data used in the experiments is a publicly available training dataset at Nasa's website (`https://data.nasa.gov/Aerospace/CMAPSS-Jet-Engine-Simulated-Data/ff5v-kuh6/`). It comprises a simulated dataset containing several jet engines (turbines) sensor data. The information on the types of sensors is not shown, but it is possible to verify that they differ, as the sensors' ranges are very different.

There are eight files containing data. Four train files (train_FD0001 /0002 /0003 /0004.txt) and four tests (test_FD0001 /0002/ 0003/ 0004.txt). Table 3.1 shows how the data looks on sup-

plied files. The raw files are text-type files containing several rows and 26 unlabeled columns separated by spaces. The data was truncated here for the sake of visualization simplicity.

Table 3.1: Extract of the dataset raw file

1	1	−0.0007	−0.0004	100.0	518.67	641.82	1589.70	1400.60	14.62	21.61
1	2	0.0019	−0.0003	100.0	518.67	642.15	1591.82	1403.14	14.62	21.61
1	3	−0.0043	0.0003	100.0	518.67	642.35	1587.99	1404.20	14.62	21.61
1	4	0.0007	0.0000	100.0	518.67	642.35	1582.79	1401.87	14.62	21.61
1	5	−0.0019	−0.0002	100.0	518.67	642.37	1582.85	1406.22	14.62	21.61
1	6	−0.0043	−0.0001	100.0	518.67	642.10	1584.47	1398.37	14.62	21.61
1	7	0.0010	0.0001	100.0	518.67	642.48	1592.32	1397.77	14.62	21.61

The raw datasets contain the turbine identification number (first column) the number of running cycles for each row (second column) and three operational settings (third to fifth columns). The remaining 21 columns contain the data for each sensor.

At the train files, the machines are considered to be operating normally in the first cycle, and at some unknown cycle for each equipment, the failure starts developing, until it fails, at the last cycle. In the test files, the series ends some time prior the failure. As in the current work, the idea is to check how machine learning would predict the actual state of the machine (between three health levels) until it fails. The test files were not used.

The files with indexes 0001 and 0002, contain only one fault modes, while the 0003 and 0004 contain two fault modes. The developed fault modes are not identified per machine in the second case. As the training methods required labels to identify different failure modes, only single failure mode data was used.

Chapter 4

Proposed Method

To assess the usage of federated learning for remaining useful life predictions, two Pyhton scripts were created. The first one implements the server for the FL method, and the second one the clients. The simulations were run using only one computer using five Python parallel instances. In a "production" implementation, multiple computers would be used, one being the server and the others being the clients. Each of the clients only access to its own data and exchange training parameters to and from the server. Figure 4.1 shows the Python instances structure.

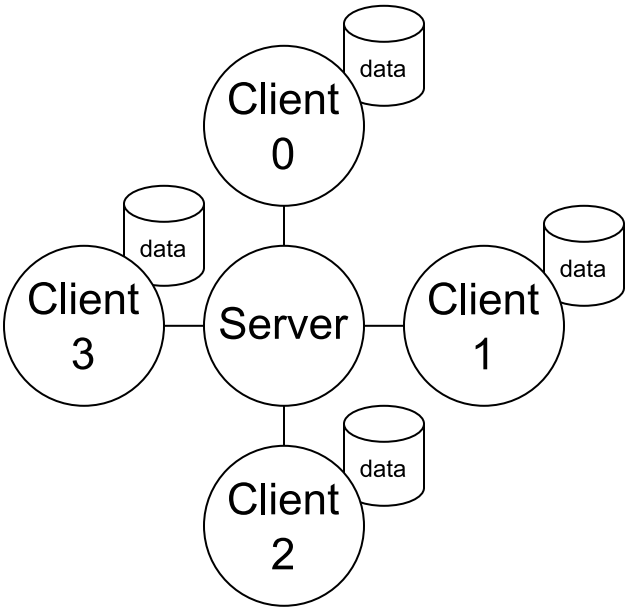


Figure 4.1: Federated Learning schema

Multiple simulations are run just by changing the configuration of the neural networks. The number of layers and elements per layer may be changed. These parameters are changed on the file *params.json*. One example of the file can be found in Appendix A.1.

4.1 Data Intake

The original data is available in different text files containing a list of values separated by space. The data consists of the following:

- Machine id: identification of each piece of equipment.
- Cycle: running cycle from one to the last cycle before failure.
- Settings: Three operational setting parameters.
- Sensors: Twenty-one sensors.

There is no given information of the types of sensors. Also there is no given information on their relevance to the failure mode of the equipment. The cycle order was changed on the original files for convenience. Instead of representing the number of running cycles, they represent Cycles to failure. Cycle *zero* is then the cycle where the equipment fails. There is data for a total number of one hundred turbines. Table 4.1 shows how the data is organized on the given files and how the cycles to failure were obtained from the original running cycles.

Table 4.1: Extract of the dataset, showing “Cycles” to “Cycles to failure” change.

Machine id	**Original Cycles**	**Cycles to failure**	**Setting 1**	**Setting 2**	**Sensor 1**	**Sensor 2**
1	1	191	-0.0007	-0.0004	641.82	1589.7
1	2	190	0.0019	-0.0003	642.15	1591.82
1	3	189	-0.0043	0.0003	642.35	1587.99
1	...	...	...	...	...	...
1	...	...	...	...	...	...
1	190	2	-0.0019	-0.0001	643.64	1599.22
1	191	1	0.0009	0.0001	643.34	1602.36
1	192	0	-0.0018	0.0003	643.54	1601.41

4.2 Health status

During runtime, the cycles to failure are converted to health statuses to be used as a reference for the training. They are converted to three columns of data containing values *zero* or *one*.

Table 4.2: Cycle to failures conversion to health states

Condition	Healthy	Alert	Danger
$CF > 50$	1	0	0
$15 < CF \leq 50$	0	1	0
$CF \leq 15$	0	0	1

Table 4.2 shows the conditions to define each of the columns of the states, where CF is each data row cycle to failure. Figure 4.2 is a representation of the life cycle of the equipment. Each machine may contain any number of "Healthy" rows, but always the last fifty rows are considered "Alert" or "Danger".

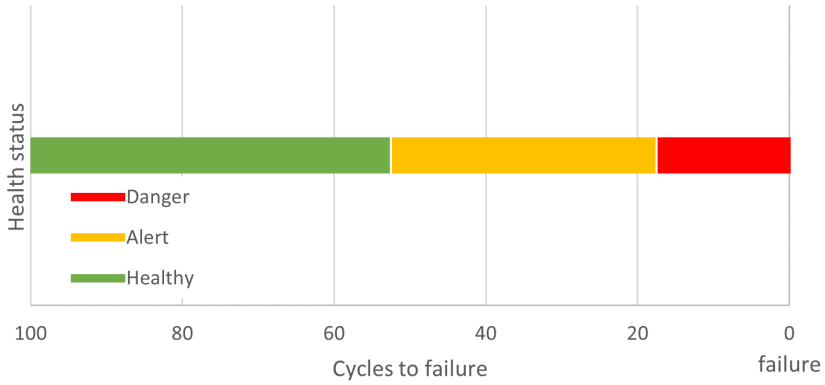


Figure 4.2: Plot representation of the possible health states

The objective of this change from "cycles to failure" to "health status" is to turn the problem into a classification one instead of trying to predict the cycles to failure in a kind of regression method. This conversion brought some benefits.

- Regression in not necessarily monotonous. Results may oscillate, meaning the number of cycles to failure may increase instead of decrease.
- The classification outputs three values of the probabilities of the machine to be in each of the three states.
- The classification was found to be more stable, although there are cases where the equipment may go from "Alert" back to "Healthy". It happened rarely.
- It is possible to have an estimate of the failure distribution once in the "Danger" state based on the historical data, as shown in Figure 2.3

4.3 Sensors Filtering

As the sensors' original data is very noisy, the Savitzky-Golay filter was applied to smooth the signals. It is possible to apply the filter a predefined number of times to achieve more noise removal. The script used for the filtering is the function *sg_filter* shown in Appendix A.2.

Figures 4.3 and 4.4 show Sensor 3 data before and after filtering. The filter was applied three times to obtain this filtered data. The different colors refer to different machines' data. As it is possible to observe on the plots, the noise was filtered out, but the main trends, which are important for the predictions, were kept. The filter is applied before training and running the predictions.

The best-obtained filtration parameters were found to be:

- *window_lenght* = *19* and *polyorder* = *1*, which is actually a moving average;
- *window_lenght* = *35* and *polyorder* = *2*;

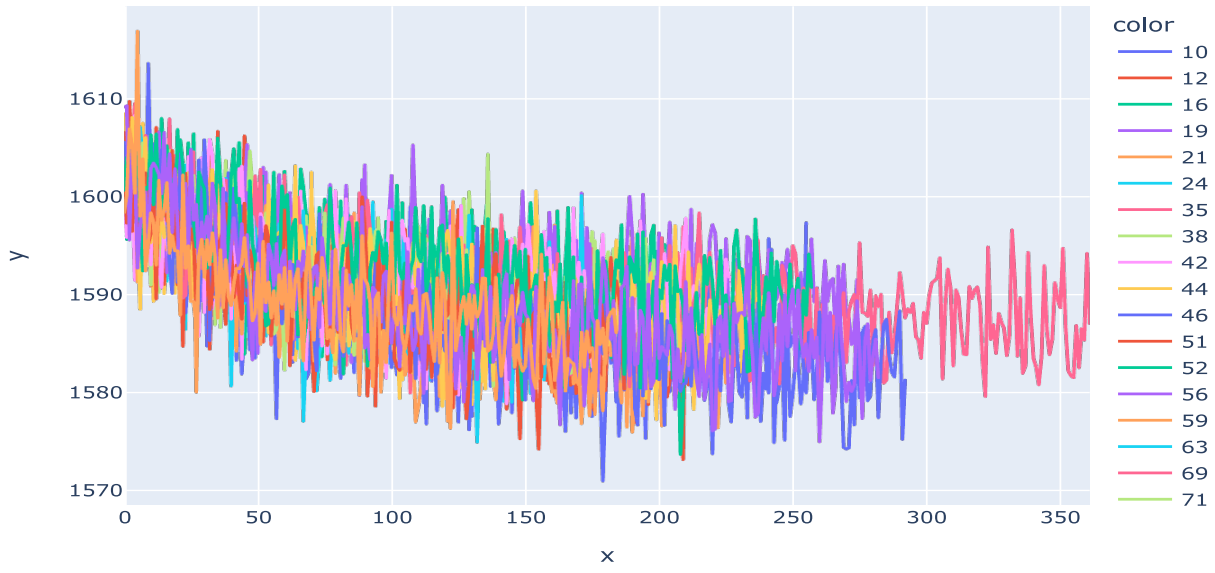


Figure 4.3: Sensor 3 data plots before filtering

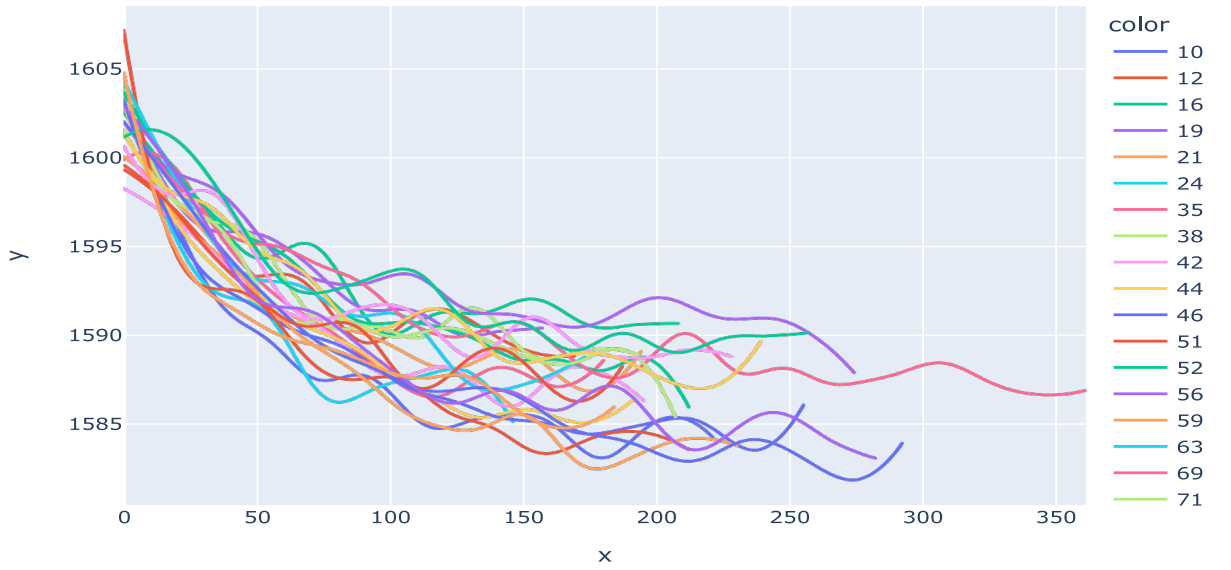


Figure 4.4: Sensor 3 data plots after filtering

4.4 Sensors Selection

The selection of sensors in use in the machine learning process was made using the Spearman's rank coefficient. This was the chosen method because it does not assume that the correlations are linear. The script used for the calculations and plots can be found in Appendix A.2.

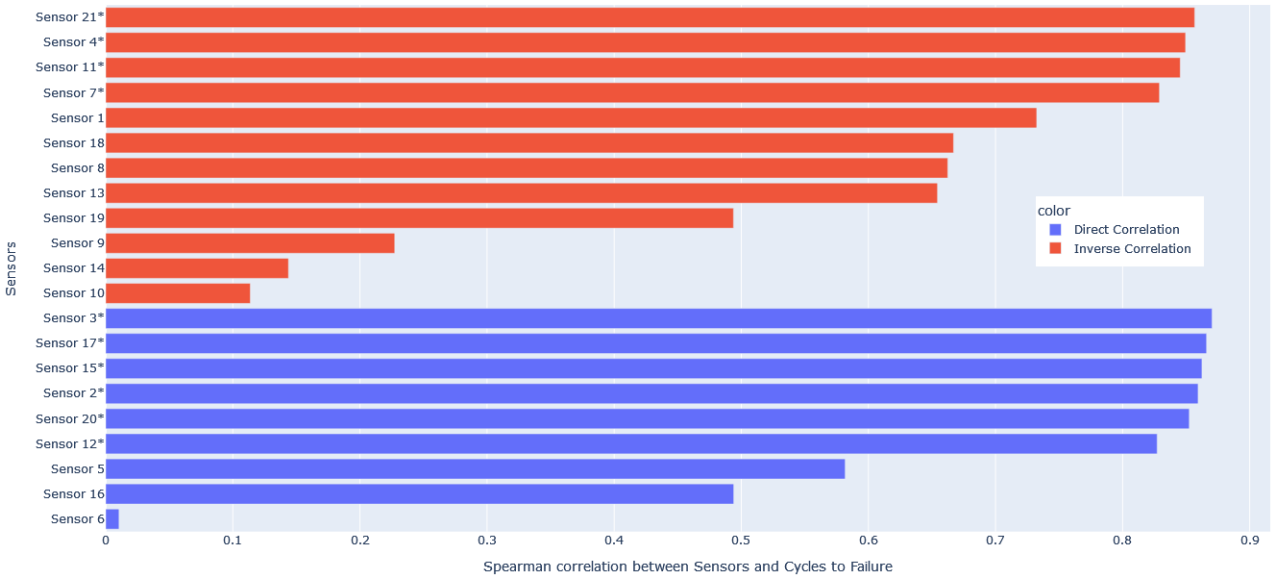


Figure 4.5: Sensors Spearman's correlations

Figure 4.5 shows the results of the correlations between each sensor and the machines' cycles to failure. The colors of the bars show the ones with direct (positive) and inverse (negative) correlations.

The sensors which correlate above 0.8 were selected for machine learning. These sensors are marked with an asterisk on the plot. The remaining sensors were ignored in the machine learning process.

4.5 Server

The server script defines all the procedures associated with it. The basic functions are detailed:

- Randomically sort the machines to be used on each client.
- Save one file per client containing each dataset.
- Start the server and wait for clients to connect.
- Run the federated learning routine by receiving training results from clients and sharing the combined parameters with the clients.
- Stop the server.
- Collect the client output files containing the results of the predictions for this run and join them for report.

This cycle is repeated for the defined *repetitions* variable and also repeated for the amount of different neural network parameters. After all the cycles, plots are created for further analysis. The server script is presented in Appendix A.3. The number of clients should be set as a *clients* variable at the script. The server waits for all clients to connect before starting the training sessions.

4.6 Client

The client script defines all the procedures associated with it. This is the same script for all the clients. The variable *client* holds an integer number defining the instance it should be. The basic functions are detailed:

- Read the data.
- Connect to the server.

- Execute preliminary training for initial weights and thresholds.
- Run the federated routine: Testing the received parameters with the next dataset; Train on this dataset; Share parameters back to the server.
- Create the output files containing the results of the predictions for this run.
- Disconnect from the server and wait for it to prepare new datasets.

Appendix A.4 contains the code for the clients' script. All the clients use identical scripts. The only difference between them is the variable *client* that needs to be set for each case. This project used four clients, so the values used for each client were *0, 1, 2, 3*. This is needed because the server stores files with different indexes addressed to each client during the preparation phase. And at the end of each cycle, each client stores its results back to a file to be read by the server.

The main part of the client script is the definition of the *Client* class. It defines how the client will cyclically train and test the obtained parameters. The package *Flower* (`https://flower.dev/`) controls the federated learning process. It handles the connections between the clients and the server and the parameters sharing between them. The package *Tensorflow* (`https://www.tensorflow.org/`) is used for machine learning-related processes. It handles training and predictions.

4.7 Process Flow

The process was split into four basic steps, as shown in Figure 4.6. At flow preparation, the neural network (NN) parameters are read. They define the number of layers and a number of elements in each layer. As there are actually three outputs, being the probabilities of the machine being in each status ("Healthy", "Alert" and "Danger"), the last layer necessarily contains three elements. A total of three and four layers was tested. For each layer, the number of elements was changed from the values in the group [4, 16, 32, 64]. These parameters are set on the file *params.json*; one example can be seen in Appendix A.1.

The flow contains two main loops to process all the simulations. *Loop 1* is repeated a number of times with the same NN parameters. The objective of this loop is to have more samples and to get a better estimate of the number of cycles each status contains, as a Monte Carlo simulation. The *repetitions* variable defines the number of times it will run, and it should set the same value for the server's and clients' scripts. The function of *Loop 2* is to cycle through all the defined NN parameters.

Ultimately, the scripts should run "the total number of NN parameters" times "the defined running repetitions". This means that the number of times to run can easily reach the order of hundreds, and the scripts may take a long time to complete. Also the computer's memory may

also impose a limitation on the number of cycles as its usage increases after each repetition. In this case, the number of parameters and repetitions should be controlled.

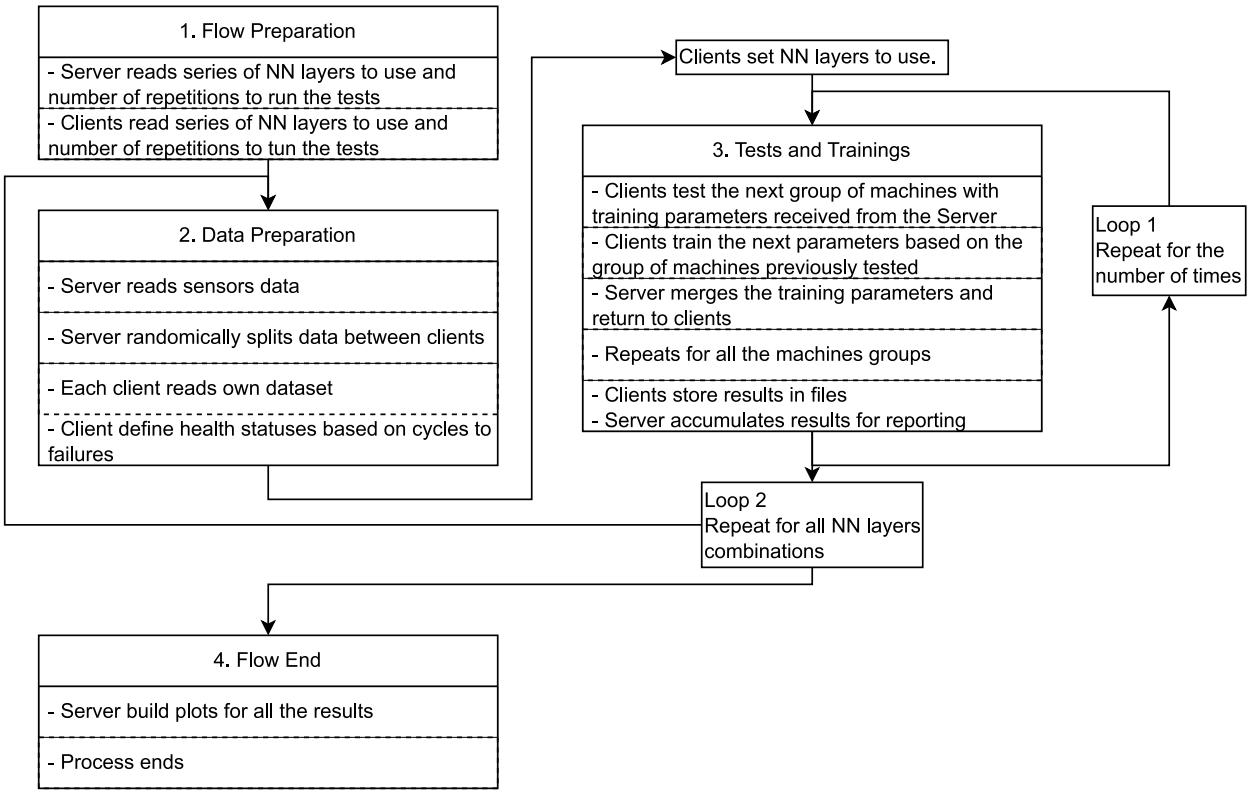


Figure 4.6: Process flow for the simulations

4.8 Simulations Results

Every remaining useful life prediction (as any machine learning prediction) has an uncertainty associated to it. Suppose there is an estimate of the probability of a remaining useful life prediction being correct and the costs related to either the failure or maintenance. In that case, it is possible to decide based on them. Figure 4.7 shows the results of several simulations of a system trained to predict a failure 15 cycles before they happen (Danger state). The interpretation of the plot follows. Given that a prediction of 15 cycles to failure was made, the probability of the equipment surviving the exact number of cycles is given by the blue bars. Given the same prediction, orange curve gives the probability of it surviving the exact number of cycles or less. Cycles in this case mean an operation cycle or “run” of a machine, for example, an airplane flight.

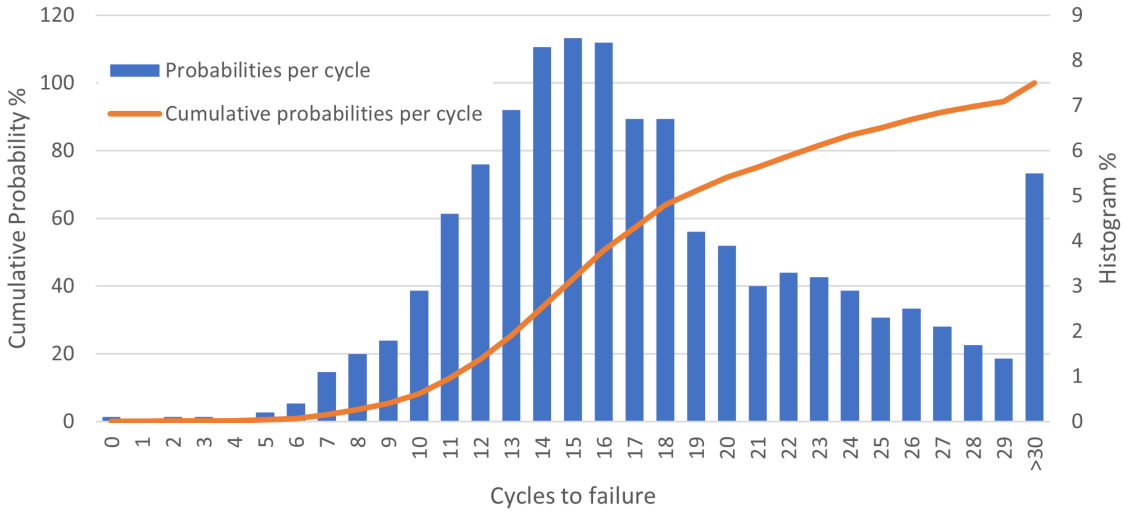


Figure 4.7: Probabilities of failure in the next cycles

The histogram plot shows, for example, that having 15 cycles RUL prediction:

- There is a probability of 8,5% of exactly 15 cycles to failure.
- There is a probability of 1% of having actually 8 cycles before failure.

Usually the cumulative probability brings more relevant data about ranges, and this may be useful for decisions on how to act as for example:

- There is a probability of 80% of the equipment failing with less than 23 cycles.
- Or 90% probability of failure in the next 26 cycles.
- There is nearly 99% chance of the failure not happening in the next 6 cycles.

If it is considered important to build a system to detect future failures, this equipment has a relevant failure cost. This cost may be directly related to its unavailability (i.e. production loss), or the maintenance cost may be higher in case of failure, safety risks.

If a chance of failure of 20% is accepted, (or to avoid 80% of them), ideally the maintenance or a replacement should be done before 12 cycles after the danger state detection.

Chapter 5

Discussion and Conclusions

This Chapter contains the discussions about the simulations and their results. Some dificulties in running the simulations are included to avoid repetitions in case of further work.

5.1 Simulation Process

The simulations were run on a "home" computer with limited resources. Although it was able to run all the simulations, the simulations took a long time to complete with occasional crashes forcing the simulations to be restarted.

A "lightweight" Linux distribution was used to run the simulations because the operational system consumes significantly less memory and processing power than other systems.

5.1.1 Running Times

The training was run using four clients and the server instances in parallel. This means that the computer resources were split between them. The server running in parallel with the clients is not expected to cause delays since its processing (mainly the merging of all the training parameters) is run while the clients "wait" between two training sessions on the clients' side. The clients running in parallel share all processing power between them, and it is expected to slow down the process, when compared with single-script training.

There is the possibility to run training using a GPU (graphics processing unit) instead of the computer's main processing unit. The calculations are expected to be much faster, but as this requires specific hardware, it was not tried during this work. Of course, in a real-life federated learning each client only needs to process its data. So no parallel processing would be needed having its own isolated device.

5.1.2 Memory Crashes

It was observed that after each training cycle, memory usage increases. If too many cycles are programmed to be tested, the memory reaches its limits and the system crashes. This issue was solved by limiting the number of neural networks to test at each session, making *params.json* with fewer parameters set and adjusting the number of repetitions per parameter. Appendix A.1 shows one example of three parameters, but eight parameters should run safely for ten repetitions at an 8Gb memory machine running a "lightweight" operational system.

5.2 Comparison of Neural Networks Parameters

Neural networks have some parameters that need to be provided for the machine learning method. They are the number of layers, the number of neurons, and their types per layer.

The type of neurons used was ReLU (rectified linear unit) for all layers but the last one. As the output is expected to be the probabilities of each of the three health states, three softmax elements were used as the last layer for all the cases. Each neuron in this layer outputs the probability of the equipment being at one of the possible states.

All the combinations of 4, 8, 16, 32 and 64 elements per layer were tested with a total of three and four layers (including the output) with five simulations per combination. The combinations with the best results were used again with ten simulations.

Figures 5.1 and 5.2 show two considered "bad" results compared to figures 5.3 and 5.4. All the figures showed the histogram of the remaining useful life when the equipment was predicted to be in the "Dangerous" state. Ideally, this should happen 15 cycles before the failure. As a consequence, a well-trained neural network should present a peak around 15 cycles and low dispersion around it. Figure 5.1 shows an important dispersion, having some "Danger" predictions above 60 cycles before failure. Figure 5.2 shows a slightly lower dispersion, but an increased peak at $zero$ to 2 buckets. This means that many machines could have no warning before failure.

Figures 5.3 and 5.4 present results of much better neural network configurations. The dispersion is much lower than the other cases, with no cases above 45 cycles to failure on "Danger" detection, and a probability smaller than 0,1% of no warning before failure.

Normally neural networks require these testing steps. Different applications and data may have different sets of "best" parameters. Not necessarily a higher number of neurons obtain better results.

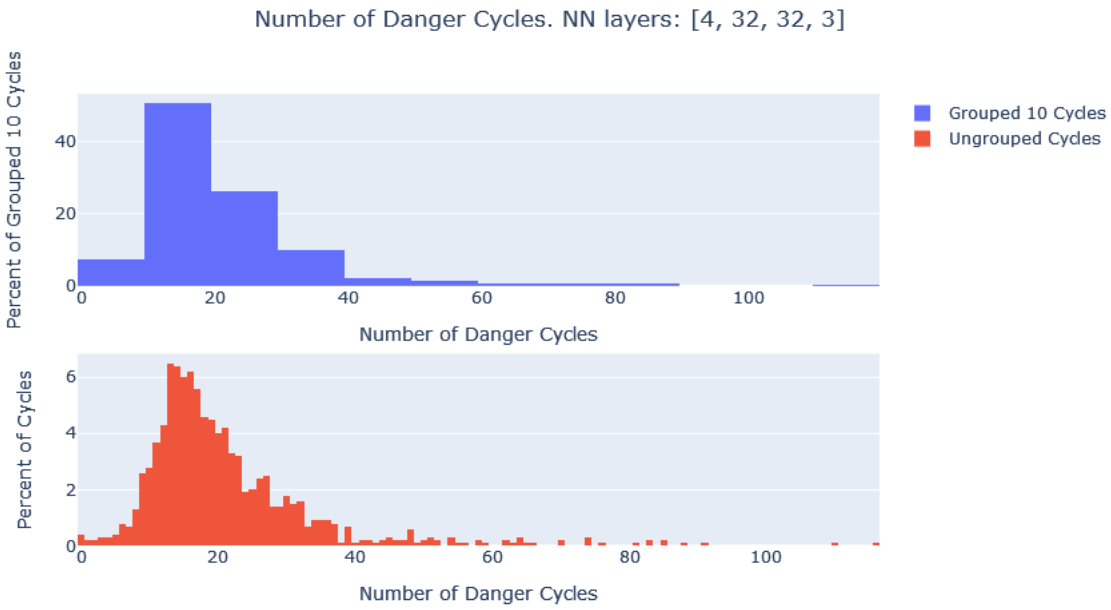


Figure 5.1: Histogram of a considered “bad” NN combination [4, 32, 32, 3] layers

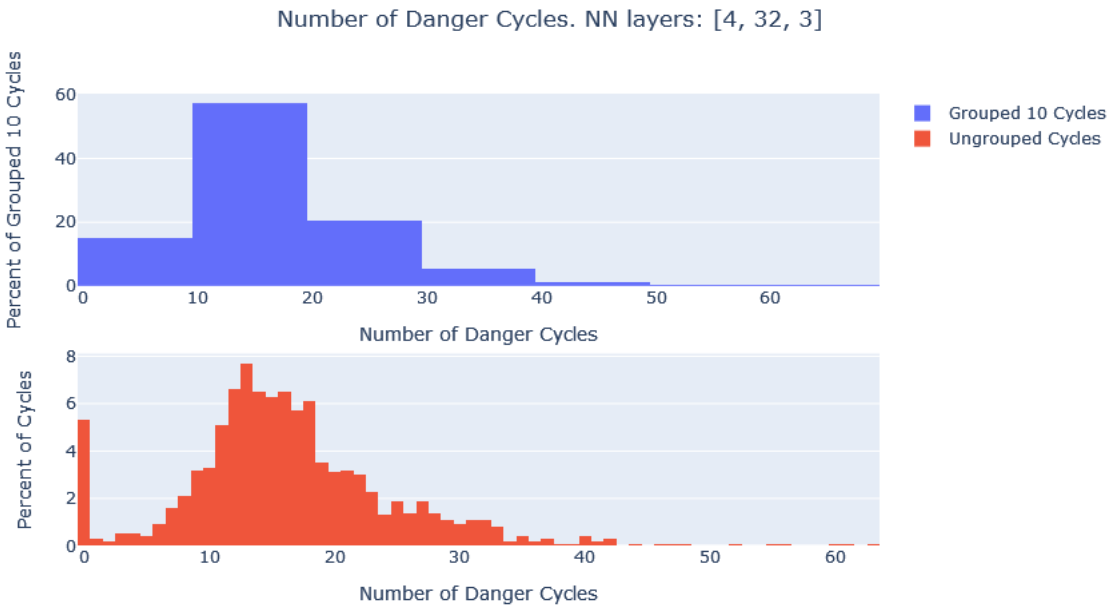


Figure 5.2: Histogram of a considered “bad” NN combination [4, 32, 3] layers

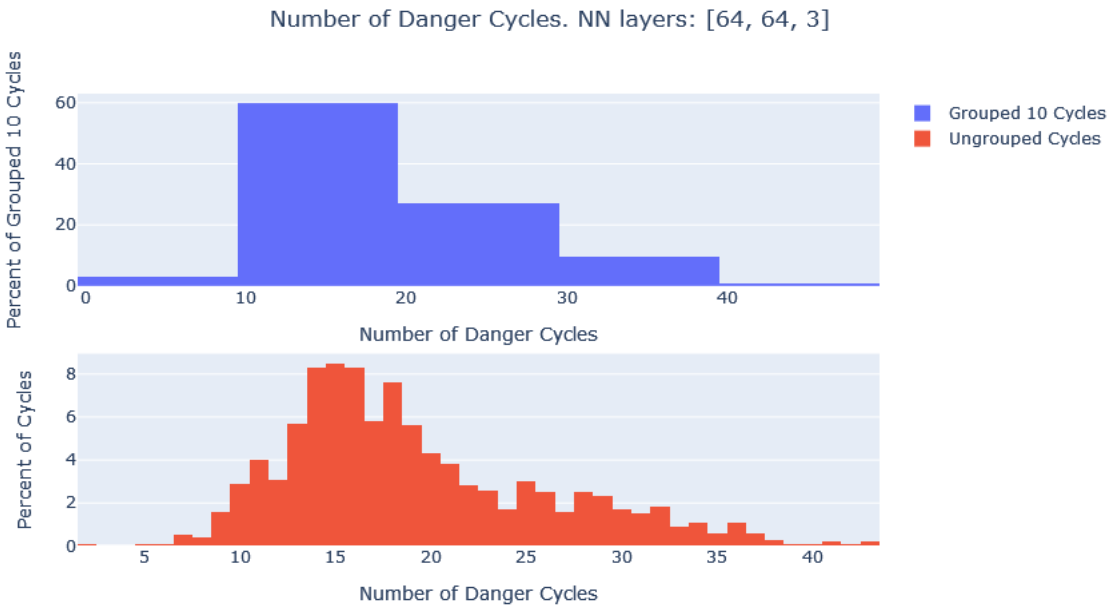


Figure 5.3: Histogram of a considered “good” NN combination [64, 64, 3] layers

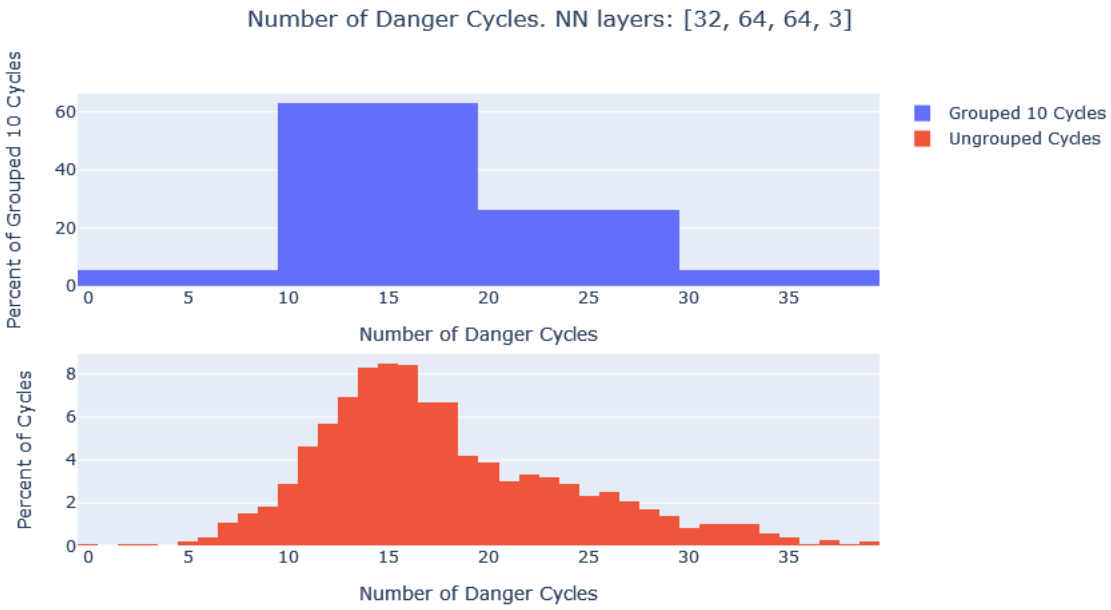


Figure 5.4: Histogram of a considered ”good” NN combination [32, 64, 64, 3] layers

5.3 Results

One set of plots was generated for each used neural network parameter containing one plot per machine. The plot in Figure 5.5 is an example of a desirable result. The blue line represents the observed cycles to failure. The yellow and red dots pinpoint the start of the desired alert and danger detections. The shaded areas display the most likely state the machine learning model predicted. In a perfect prediction, the dots should coincide with the start of the area with the same color.

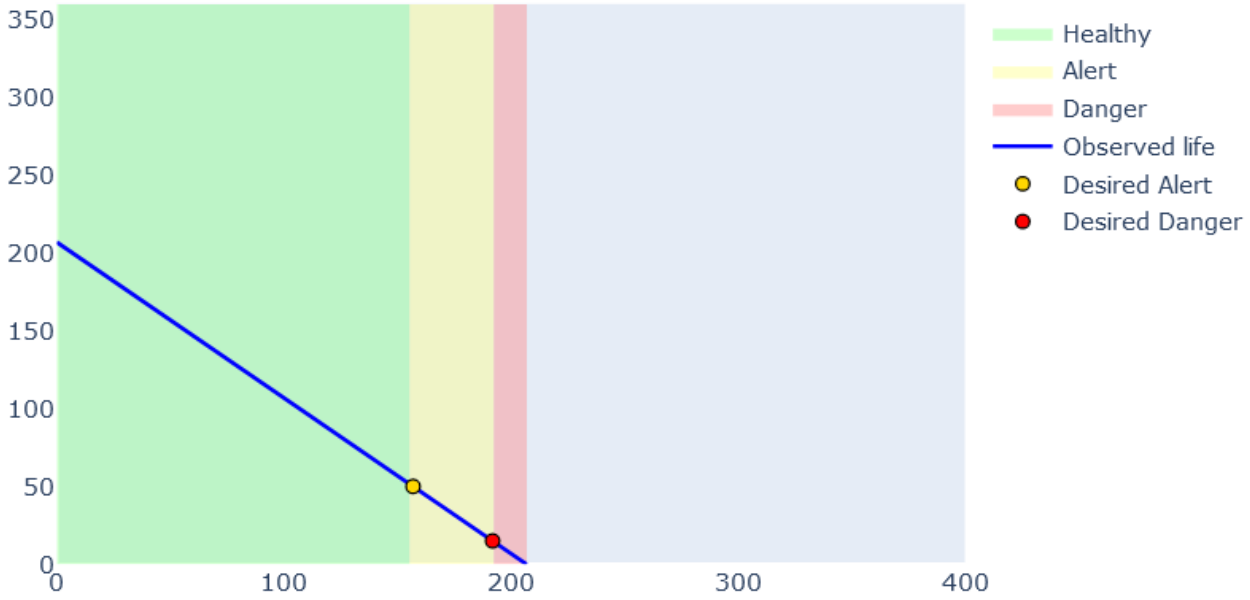


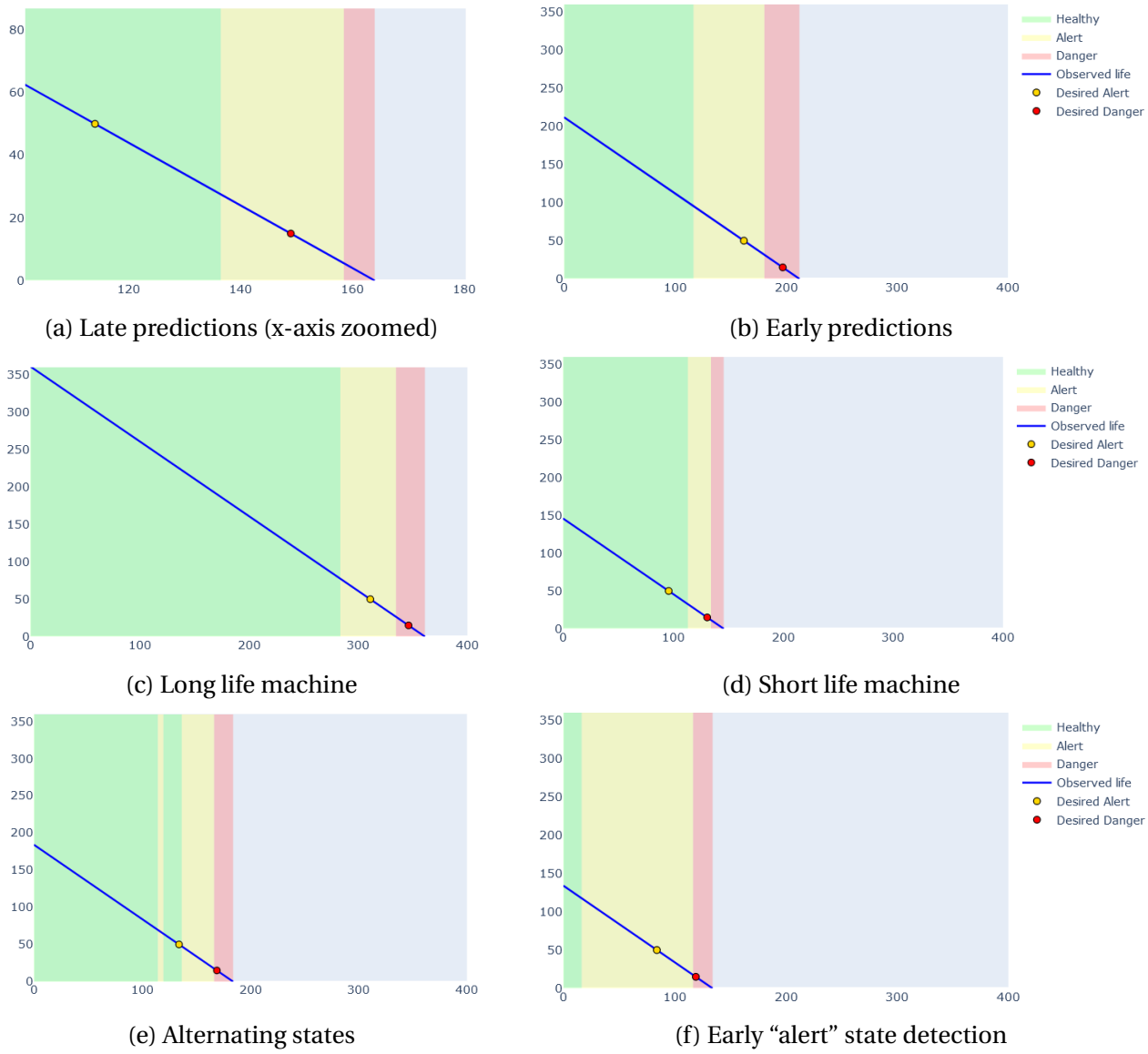
Figure 5.5: Desirable prediction

The plots in Figure 5.6 show results for some selected machines that presented different outcomes, which are worth mentioning. Figure 5.6a show late predictions while figure 5.6b shows early predictions. In rare cases (0.1% of them) there was no “danger” prediction before the failure occurred for these parameters. Figure 5.6e shows one case where some cycles were detected as “alert” states with posterior return to “healthy” ones. This kind of event events was not very frequent (around 3% of the cases). No similar case involving a “danger” detection was found.

One concern during the tests was how the predictions would behave in the case of machines with longer or shorter lives compared to the average life duration. Figures 5.6c and 5.6d show a very long life with more than 350 cycles to failure, and a very short life, with less than 150. Although the predictions were a bit early and late for the long and the short life-lasting machines, respectively, they similar to the ones closer to the average cycles to failure.

The plots shown in Figure 5.6 are from a particular run for the artificial neural network parameters [32, 64, 64, 3]. As the sequence of machines was changed at each run, the results dif-

fered at each time. The trained detection of the “alert” state was 50 and the “danger” was 15 cycles to failure.



(a) Late predictions (x-axis zoomed)

(b) Early predictions

(c) Long life machine

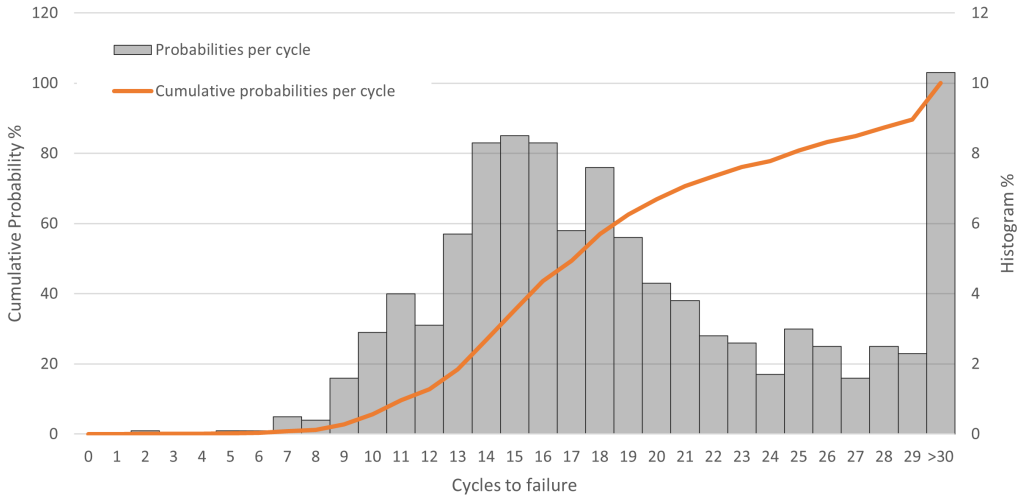
(d) Short life machine

(e) Alternating states

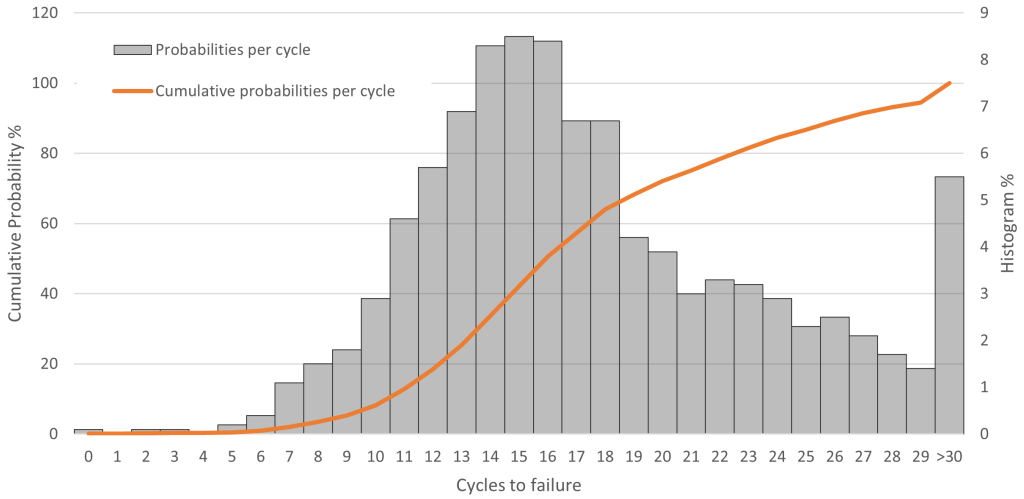
(f) Early “alert” state detection

Figure 5.6: Results examples [32, 64, 64, 3]. x-axis: running cycles; y-axis: cycles to failure

The plots in Figure 5.7 show the histogram and cumulative plots of the results of two neural networks ([64, 64, 3] and [32, 64, 64, 3]). The results show that if, according to cost calculations shown in the Section 2.1.2, it is acceptable to have an 80% chance of repairing before the equipment fails, it would be necessary to make a maintenance or component replacement 12 cycles after the “danger” detection. Making the maintenance 10 cycles after the detection would result in a probability of around 90% of the equipment not failing before it.



(a) Neural network parameters [64, 64, 3]



(b) Neural network parameters [32, 64, 64, 3]

Figure 5.7: Histogram and cumulative plots for different neural networks

5.4 Predictive and Age-Based Maintenance Comparison

Similarly to the previous ones, figure 5.8 shows a histogram and cumulative probability, but now for the total life of all the machines. The width of the bars corresponds to buckets of 12 cycles and their height represents the number of machines that failed in that cycles period. With this kind of information, if 20% were to be considered as the maximum probability of an equipment failure, it would be expected to conclude that preventive maintenance should be done after around 160 cycles (age-based time t_{AB}). This is a simple approach to age-based maintenance.

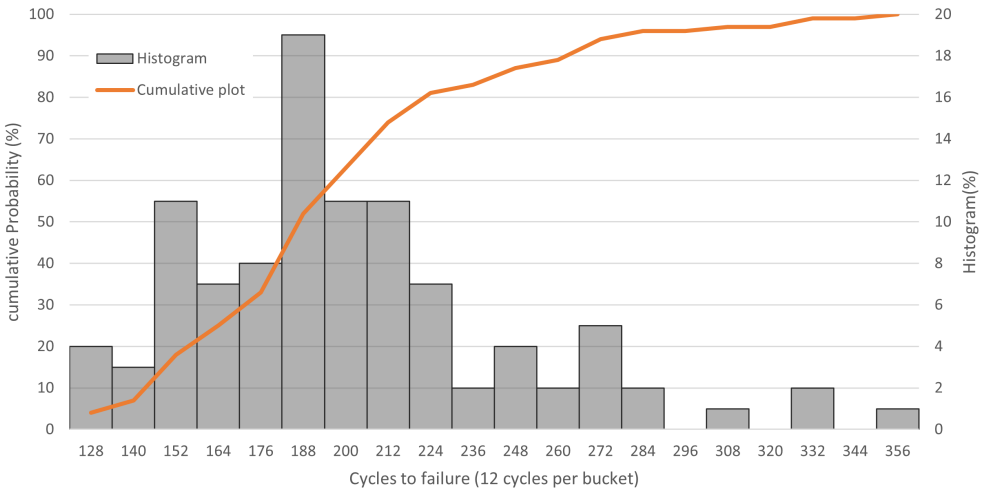


Figure 5.8: Histogram and cumulative plot for total cycles to failure

Considering the cumulative probability shown in Figure 5.7b, it may be concluded that estimating the last cycle to make maintenance is possible given a “danger” detection and a maximum failure probability allowed. Also considering that the maximum chance of failure should be 20%, a maintenance should be done at most around 12 cycles after the prediction (predictive time t_{PD}).

Making age-based decisions instead of using a machine learning approach would be equivalent to considering that all the machines are expected to fail at 172 cycles ($t_{AB} + t_{PD} = 160 + 12$). At the same time, nearly 50% of them could survive between 200 and 300 cycles. The maintenance could then be delayed, saving both repair and lost profits due to unavailability. As would be expected, the uncertainty decreases when using a machine learning approach.

These results are not exactly due to federated learning, but the machine learning-based predictive maintenance. But since a big pool of similar equipment is needed to improve the results, a small industry could only have data for reliable predictions with FL. Small industries could then benefit from partnering with other companies without sharing raw data. The results provided indicate that this approach is possible.

5.5 Conclusions

The results of the simulations have demonstrated that the proposed method is viable for a predictive maintenance strategy based on remaining useful life estimation using federated learning and artificial neural networks. By using distributed computing and machine learning, failure prediction models can be trained using data from multiple locations. This allows different companies to cooperate, building a better training for more accurate predictions without sharing original data.

In an industrial system, predictive maintenance improves reliability by reducing the downtime and increasing overall efficiency while decreasing the costs.

Although the ability of artificial neural networks to learn how to predict complex problems is remarkable, other machine learning methods may be advantageous and further testing may bring valuable information. Certainly there is space for further studies, improvements and tests with online plant data. Specific techniques for enhancing privacy in federated learning systems are available in the literature that could be expanded for this application.

Federated learning has significant potential due to the possibility of aggregating information on the volume of data needed for acceptable training and its ability to deal with data privacy concerns, GDPR and other regulations.

Chapter 6

Ideas for Further Work

In summary, this project aims to conduct a rigorous evaluation of a specific implementation of federated learning in conjunction with an artificial neural network in a controlled setting to determine its viability. Although it is not this study's central objective, investigating this model's performance in real-world scenarios and implementing online feedback for the equipment operator would be a fascinating avenue for future research.

- Simulation with real data instead of synthetic data
- Real federated learning running on multiple devices with real and live data
- Asynchronous training (only one device adding information to training instead of the whole group)
- Multiple identified failure modes
- Try diverse machine learning methods

Hoping for further work, the codes are made available in the appendix section.

Bibliography

Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A., and Seth, K. (2017). Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1175–1191, New York, NY, USA. Association for Computing Machinery.

Du, N. H., Long, N. H., Ha, K. N., Hoang, N. V., Huong, T. T., and Tran, K. P. (2023). Trans-lighter: A light-weight federated learning-based architecture for remaining useful lifetime prediction. *Computers in industry*, 148.

Guo, L., Yu, Y., Qian, M., Zhang, R., Gao, H., and Cheng, Z. (2022). Fedrul: A new federated learning method for edge-cloud collaboration based remaining useful life prediction of machines. *IEEE/ASME Transactions on Mechatronics*, pages 1–10.

Hard, A., Rao, K., Mathews, R., Beaufays, F., Augenstein, S., Eichner, H., Kiddon, C., and Ramage, D. (2018). Federated learning for mobile keyboard prediction. *CoRR*, abs/1811.03604.

Lettvin, J. Y., Maturana, H. R., McCulloch, W. S., and Pitts, W. H. (1959). What the frog's eye tells the frog's brain. *Proceedings of the IRE*, 47(11):1940–1951.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133.

Missura, O. and Gaertner, T. (2009). Player modeling for intelligent difficulty adjustment. In Gama, J., Costa, V., Jorge, A., and Brazdil, P., editors, *DISCOVERY SCIENCE, PROCEEDINGS*, volume 5808 of *Lecture Notes in Artificial Intelligence*, pages 197–211. Univ PORTO; LIAADO; SAS; CRACS InescPorto LA; YAHOO. 12th International Conference on Discovery Science, Porto, PORTUGAL, OCT 03-05, 2009.

Nielsen, M. (2019). *Neural Networks and Deep Learning*.

Rausand, M., Barros, A., and Hoyland, A. (2021). *System Reliability Theory: Models, Statistical Methods, and Applications*. Wiley Series in Probability and Statistics. Wiley, 3rd edition.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Savitzky, A. and Golay, M. J. E. (1964). Smoothing and differentiation of data by simplified least squares procedures. *Analytical chemistry (Washington)*, 36(8):1627–1639.

Yue, W., Wang, Z., Chen, H., Payne, A., and Liu, X. (2018). Machine learning with applications in breast cancer diagnosis and prognosis. *Designs*, 2(2).

Zhang, C., Xie, Y., Bai, H., Yu, B., Li, W., and Gao, Y. (2021). A survey on federated learning. *Knowledge-Based Systems*, 216:106775.

Appendix A

Scripts

Here the scripts are shared for replication of the method, and hopefully further improvement. Comments are included in the hope of making it easier to understand.

A.1 Parameters file example (*params.json*)

```
[[64, 64, 3],
[16, 64, 4, 3],
[32, 32, 32, 3]]
```

A.2 Sensor check script

```
# -*- coding: utf-8 -*-
"""
Created on Sat Mar 18 11:41:13 2023

@author: lflsa
"""
import pandas as pd
import json
import scipy.signal as ss
import plotly.express as px


def sg_filter(df, times):
    for i in range(times):
```

```
        sensors = df.columns.to_list()[2:]
        for unit in df["Machine"].unique():
            for sensor in sensors:
                df.loc[df["Machine"] == unit, sensor] = ss.savgol_filter(
                    df.loc[df["Machine"] == unit, sensor],
                    window_length=35,
                    polyorder=2,
                    deriv=0,
                    mode="interp",
                )
    return df


file_train = "FD001/train_FD001_cycle_to_failure.txt"

column_names = ["Machine", "Cycles", "Setting_1", "Setting_2", "Setting_3"]
for i in range(21):
    column_names.append("Sensor_" + str(i + 1))
full_train_data = pd.read_csv(file_train, names=column_names, delimiter="\t")
data = sg_filter(full_train_data, 3)
data = data[data["Cycles"] <= 100]

correlations = []
sensors = []
pos_neg = []
for column in column_names:
    if "Sensor" in column:
        print(
            "Correlation_"
            + column
            + ":_"
            + str(data["Cycles"].corr(data[column], method="spearman"))
        )
        correlations.append(abs(data["Cycles"].corr(data[column], method="spearman")))
        pos_neg.append(data["Cycles"].corr(data[column], method="spearman"))
        if correlations[-1] >= 0.8:
            sensors.append(column + "*")
```

```
        else:
            sensors.append(column)
        sensors_ordered = [x for _, x in sorted(zip(correlations, sensors))]
        correlations_ordered = correlations.copy()
        correlations_ordered.sort()
        pos_neg_ordered = [
            "Inverse Correlation" if x < 0 else "Direct Correlation" for x in pos_neg
        ]


fig = px.bar(
    x=correlations_ordered,
    y=sensors_ordered,
    color=pos_neg_ordered,
    orientation="h",
    text_auto=False,
)
fig.update_layout(
    xaxis_title="Spearman correlation between Sensors and Cycles to Failure",
    yaxis_title="Sensors",
)
fig.write_html("correlations.html")
```

A.3 Server Script

```
"""
@author: lflsa
"""
import time
import pandas as pd
import flwr as fl
import plotly.graph_objects as go
import numpy as np
from datetime import datetime
import json
import os
from statistics import mean, stdev
```

```
import random

import plotly.express as px
from plotly.subplots import make_subplots

b1 = 50  # Threshold for alert level
b0 = 15  # Threshold for danger level
repetitions = 10  # Number of times to test each parameter
clients = 4  # Minimum number of clients

####
# Implements Flower server
####
def startServer(rounds):

    strategy = fl.server.strategy.FedAvg(
        min_available_clients=clients,
    )

    fl.server.start_server(
        server_address="0.0.0.0:8080",
        config=fl.server.ServerConfig(num_rounds=rounds),
        strategy=strategy,
    )



####
#   Reads the Neural Netork parameters to be tested
####
def read_params():
    if os.path.isfile("params.json"):
        with open("params.json") as target:
            params = json.load(target)

        return params
```

```
####
#   Reads the data as a dataframe, adding the columns names
####
def read_data():
    removetags = [
        "Setting 1",
        "Setting 2",
        "Setting 3",
        "Sensor 1",
        "Sensor 5",
        "Sensor 6",
        "Sensor 8",
        "Sensor 9",
        "Sensor 10",
        "Sensor 13",
        "Sensor 14",
        "Sensor 16",
        "Sensor 18",
        "Sensor 19",
    ]  # Remove information that is not useful

    file_train = "FD001/train_FD001 cycle to failure.txt"
    file_test = "FD001/train_FD001 cycle to failure.txt"

    column_names = ["Machine", "Cycles", "Setting 1", "Setting 2", "Setting 3"]
    for i in range(21):
        column_names.append("Sensor " + str(i + 1))
    full_train_data = pd.read_csv(file_train, names=column_names, delimiter="\t")

    full_train_data = full_train_data.drop(removetags, axis=1)
    num_train_machines = max(full_train_data["Machine"])

    full_test_data = pd.read_csv(file_test, names=column_names, delimiter="\t")
    full_test_data = full_test_data.drop(removetags, axis=1)

    return full_test_data, num_train_machines, full_train_data, full_test_data
```

```
full_test_data, num_train_machines, full_train_data, full_test_data = read_data()
params = read_params()

machines_list = full_test_data["Machine"].unique()
machines_lists = []
for repetition in range(repetitions):
    random.shuffle(machines_list)
    machines_lists.append(machines_list.copy())

num_test_machines = max(full_test_data["Machine"])

results = {}
file_client = "results/client"

#repeats once per parameter in params.json file
for param in params:
    param_txt = json.dumps(param)
    results[param_txt] = {
        "Alert average": [],
        "Alert max": [],
        "Alert min": [],
        "Alert sd": [],
        "Danger average": [],
        "Danger max": [],
        "Danger min": [],
        "Danger sd": [],
        "Correct Alert average": [],
        "Correct Danger average": [],
        "Correct Alert sd": [],
        "Correct Danger sd": [],
        "Alert Cycles": [],
        "Danger Cycles": [],
    }

    # repeats the number of defined repetitions
    for repetition in range(repetitions):
```

```
length = int(num_train_machines / clients)
for i in range(clients):
    templist = machines_lists[repetition][i * length : (i + 1) * length]
    full_train_data[full_train_data["Machine"].isin(templist)].to_csv(
        file_client + "train" + str(i) + ".csv", index=False
    )
    full_test_data[full_test_data["Machine"].isin(templist)].to_csv(
        file_client + "train" + str(i) + ".csv", index=False
    )

print("data shuffled")

startServer(5) #argument defines number of rounds per experiment
time.sleep(10) #gives time to the clients to finish
outputs = pd.DataFrame()
for client in range(clients):
    outputs = pd.concat(
        [outputs, pd.read_csv("output" + str(client) + ".csv", sep=",")]
    )
#####
# here finishes the main functions. the code that follows is for reporting pu
#####
alert = []
danger = []
correct_alert = []
correct_danger = []

for machine in outputs["Machine"].unique():
    correct_alert.append(
        outputs[outputs["Machine"] == machine]["Correct Alert"].sum()
    )
    correct_danger.append(
        outputs[outputs["Machine"] == machine]["Correct Danger"].sum()
    )
    alert.append(outputs[outputs["Machine"] == machine]["Alert"].sum())
    danger.append(outputs[outputs["Machine"] == machine]["Danger"].sum())
```

```
        results[param_txt]["Correct␣Alert␣average"].append(mean(correct_alert))
        results[param_txt]["Correct␣Danger␣average"].append(mean(correct_danger))
        results[param_txt]["Correct␣Alert␣sd"].append(stdev(correct_danger))
        results[param_txt]["Correct␣Danger␣sd"].append(stdev(correct_danger))

        results[param_txt]["Alert␣average"].append(mean(alert))
        results[param_txt]["Danger␣average"].append(mean(danger))
        results[param_txt]["Alert␣sd"].append(stdev(alert))
        results[param_txt]["Danger␣sd"].append(stdev(danger))

        results[param_txt]["Alert␣max"].append(max(alert))
        results[param_txt]["Danger␣max"].append(max(danger))
        results[param_txt]["Alert␣min"].append(min(alert))
        results[param_txt]["Danger␣min"].append(min(danger))
        results[param_txt]["Alert␣Cycles"].extend(alert)
        results[param_txt]["Danger␣Cycles"].extend(danger)

        outputs["b0"] = np.where(
            outputs["Cycles"].reset_index(drop=True) == b0, b0, None
        )
        outputs["b1"] = np.where(
            outputs["Cycles"].reset_index(drop=True) == b1, b1, None
        )

        outputs["Danger"] = outputs["Danger"] * 500
        outputs["Alert"] = outputs["Alert"] * 500
        outputs["Healthy"] = outputs["Healthy"] * 500

    data = []
    for step in np.arange(1, 101):
        data.append(
            go.Scatter(
                x=outputs[outputs["Machine"] == step]["Cycles"][::-1],
                y=outputs[outputs["Machine"] == step]["Healthy"],
                fill="tozeroy",
                fillcolor="rgba(128,255,128,0.4)",
```

```
            line_color="rgba(255,255,255,0)",
            name="Healthy",
            line={"shape": "hvh"},
            showlegend=True,
            visible=False,
        )
    )
    data.append(
        go.Scatter(
            x=outputs[outputs["Machine"] == step]["Cycles"][::-1],
            y=outputs[outputs["Machine"] == step]["Alert"],
            fill="tozeroy",
            fillcolor="rgba(255,255,128,0.4)",
            line_color="rgba(255,255,255,0)",
            name="Alert",
            line={"shape": "hvh"},
            showlegend=True,
            visible=False,
        )
    )
    data.append(
        go.Scatter(
            x=outputs[outputs["Machine"] == step]["Cycles"][::-1],
            y=outputs[outputs["Machine"] == step]["Danger"],
            fill="tozeroy",
            fillcolor="rgba(255,128,128,0.4)",
            line_color="rgba(255,255,255,0)",
            name="Danger",
            line={"shape": "hvh"},
            showlegend=True,
            visible=False,
        )
    )

    data.append(
        go.Scatter(
            y=outputs[outputs["Machine"] == step]["Cycles"],
```

```
                visible=False,
                name="Observed life",
                line_color="rgba(0,0,255,1)",
            )
        )
        data.append(
            go.Scatter(
                x=outputs[outputs["Machine"] == step]["Cycles"][::-1],
                y=outputs[outputs["Machine"] == step]["b1"],
                name="Desired Alert",
                mode="markers",
                marker=dict(color="Gold", size=7, line=dict(color="Black", width=1)),
                showlegend=True,
                visible=False,
            )
        )
        data.append(
            go.Scatter(
                x=outputs[outputs["Machine"] == step]["Cycles"][::-1],
                y=outputs[outputs["Machine"] == step]["b0"],
                name="Desired Danger",
                mode="markers",
                marker=dict(color="Red", size=7, line=dict(color="Black", width=1)),
                showlegend=True,
                visible=False,
            )
        )

    fig = go.Figure(data)

    fig.data[0].visible = True
    fig.data[1].visible = True
    fig.data[2].visible = True
    fig.data[3].visible = True
    fig.data[4].visible = True
    fig.data[5].visible = True
    # fig.data[100].visible = True
```

```
    # fig.data[200].visible = True
    # Create and add slider
    steps = []
    for i in range(len(fig.data)):
        step = dict(
            method="update",
            args=[
                {"visible": [False] * 6 * len(fig.data)},
                {"title": "Turbine: " + str(i + 1)},
            ],  # layout attribute
        )
        step["args"][0]["visible"][
            6 * (i + 1) - 6
        ] = True  # Toggle i'th trace to "visible"
        step["args"][0]["visible"][
            6 * (i + 1) - 5
        ] = True  # Toggle i'th trace to "visible"
        step["args"][0]["visible"][
            6 * (i + 1) - 4
        ] = True  # Toggle i'th trace to "visible"
        step["args"][0]["visible"][
            6 * (i + 1) - 3
        ] = True  # Toggle i'th trace to "visible"
        step["args"][0]["visible"][
            6 * (i + 1) - 2
        ] = True  # Toggle i'th trace to "visible"
        step["args"][0]["visible"][
            6 * (i + 1) - 1
        ] = True  # Toggle i'th trace to "visible"

        steps.append(step)

    sliders = [
        dict(
            active=0,
            currentvalue={"prefix": "Turbine: "},
            pad={"t": 50},
```

```
            steps=steps[0:100],
        )
    ]

    fig.update_layout(sliders=sliders)
    fig.update_xaxes(range=[0, 400], showgrid=False)
    fig.update_yaxes(range=[0, 360], showgrid=False)
    ts = datetime.now().strftime("%Y%m%d%H%M%S")
    fig.write_html("results/outputhealth " + ts + " " + str(param) + ".html")


with open("results/results " + ts + ".json", "w") as json_file:
    json.dump(results, json_file, indent=4)

for item in results:
    fig = make_subplots(rows=2, cols=1)
    fig.add_trace(
        go.Histogram(
            x=results[item]["Danger Cycles"],
            histnorm="percent",
            xbins={"size": 10},
            name="Grouped 10 Cycles",
        ),
        row=1,
        col=1,
    )
    fig.add_trace(
        go.Histogram(
            x=results[item]["Danger Cycles"],
            histnorm="percent",
            xbins={"size": 1},
            name="Grouped 2 Cycles",
        ),
        row=2,
        col=1,
    )
    fig.update_layout(
```

```
        title={"text": "Number␣of␣Danger␣Cycles.␣NN␣layers:␣" + str(item), "x": 0.5}
    )
    fig.update_layout(barmode="overlay")
    fig.update_yaxes(title_text="Percent␣of␣Grouped␣10␣Cycles", row=1, col=1)
    fig.update_yaxes(title_text="Percent␣of␣Cycles", row=2, col=1)
    fig.update_xaxes(title_text="Number␣of␣Danger␣Cycles")
    fig.write_html("results/histogram␣Danger␣" + str(item) + ".html")

    fig = make_subplots(rows=2, cols=1)
    fig.add_trace(
        go.Histogram(
            x=results[item]["Alert␣Cycles"],
            histnorm="percent",
            xbins={"size": 20},
            name="Grouped␣20␣Cycles",
        ),
        row=1,
        col=1,
    )
    fig.add_trace(
        go.Histogram(
            x=results[item]["Alert␣Cycles"],
            histnorm="percent",
            xbins={"size": 2},
            name="Grouped␣4␣Cycles",
        ),
        row=2,
        col=1,
    )
    fig.update_layout(
        title={"text": "Number␣of␣Alert␣Cycles.␣NN␣layers:␣" + str(item), "x": 0.5}
    )
    fig.update_layout(barmode="overlay")
    fig.update_yaxes(title_text="Percent␣of␣Grouped␣20␣Cycles", row=1, col=1)
    fig.update_yaxes(title_text="Percent␣of␣Grouped␣2␣Cycles", row=2, col=1)
    fig.update_xaxes(title_text="Number␣of␣Alert␣Cycles")
    fig.write_html("results/histogram␣Alert␣" + str(item) + ".html")
```

A.4 Clients script

```
# -*- coding: utf-8 -*-
"""
@author: lflsa
"""
import flwr as fl
import tensorflow as tf
import pandas as pd
import numpy as np
import plotly.express as px
from datetime import datetime
import json
import time
import os
import scipy.signal as ss

client = 0 #one file per client changing this parameter
repetitions = 10
rounds = 5


def sg_filter(df, times):

    fig = px.line(x=df["Cycles"], y=df["Sensor 3"], color=df["Machine"])
    fig.write_html("Sensor 3 before filter.html")

    for i in range(times):
        sensors = df.columns.to_list()[2:]
        for unit in df["Machine"].unique():
            for sensor in sensors:
                df.loc[df["Machine"] == unit, sensor] = ss.savgol_filter(
                    df.loc[df["Machine"] == unit, sensor],
                    window_length=35,
                    polyorder=2,
                    deriv=0,
                    mode="interp",
```

```
                )

    fig = px.line(x=df["Cycles"], y=df["Sensor␣3"], color=df["Machine"])
    fig.write_html("Sensor␣3␣after␣filter.html")

    return df


def getData(filename, max_number_cycles, mu=None, sigma=None, Ymax=None):
    training_range = 1

    data_raw = pd.read_csv(filename + "train" + str(client) + ".csv", sep=",")

    data = data_raw.copy()

    machines = data["Machine"].unique()
    Xtags = data.columns
    Xtags = [e for e in Xtags if e not in ["Machine", "Cycles"]]

    data = sg_filter(data, 5)

    data = data[data["Cycles"] <= max_number_cycles]
    data.reset_index(drop=True)

    cycles = pd.DataFrame() #define the three levels to be trained
    cycles["Danger"] = np.where(
        data["Cycles"].reset_index(drop=True) <= b0, 1, 0
    )
    cycles["Alert"] = np.where(
        data["Cycles"].reset_index(drop=True).between(b0 + 1, b1), 1, 0
    )
    cycles["Healthy"] = np.where(
        data["Cycles"].reset_index(drop=True) > b1, 1, 0
    )

    machines = data["Machine"]
```

```
    originalY = data["Cycles"].reset_index(drop=True)

    data.drop("Machine", inplace=True, axis=1)
    data.drop("Cycles", inplace=True, axis=1)

    new_params = False

    #read average and stdev of each sensor from file, so clients use the same
    #set of parameters.

    if mu is None:
        new_params = True
        avg_stdev = pd.read_csv("results/avg_stdev.csv", sep=",", index_col="index")
        mu = avg_stdev["mu"]
        sigma = avg_stdev["sigma"]

    X = (data - mu) / sigma

    if Ymax is None:
        Ymax = training_range

    Y = cycles

    X = X.reset_index(drop=True)
    Y = Y.reset_index(drop=True)
    machines = machines.reset_index(drop=True)

    if new_params:
        return X, Y, mu, sigma, Ymax, machines, originalY
    else:
        return X, Y, machines, originalY


# Define Flower client
class Client(fl.client.NumPyClient):
    def get_parameters(self, config):
        print("GET_PARAMS!!!")
```

```
        return model.get_weights()

    def fit(self, parameters, config):
        global i
        print(
            "\n\nFIT MACHINES {}!!!\n\n".format(
                machines[i * MachinesPerTrain : (i + 1) * MachinesPerTrain]
            )
        )

        x_train = x_train_total[
            train_machines_total.isin(
                machines[i * MachinesPerTrain : (i + 1) * MachinesPerTrain]
            )
        ]
        y_train = y_train_total[
            train_machines_total.isin(
                machines[i * MachinesPerTrain : (i + 1) * MachinesPerTrain]
            )
        ]

        model.set_weights(parameters)
        model.fit(x_train, y_train, epochs=100, batch_size=8, verbose=0)

        return model.get_weights(), len(x_train), {}

    def evaluate(self, parameters, config):
        def predict(x): #run prediction
            print("PREDICT!!!")
            predictions = model.predict(x)
            return predictions

        global i, predictions
        x_test = x_test_total[
            test_machines_total.isin(
                machines[i * MachinesPerTrain : (i + 1) * MachinesPerTrain]
            )
```

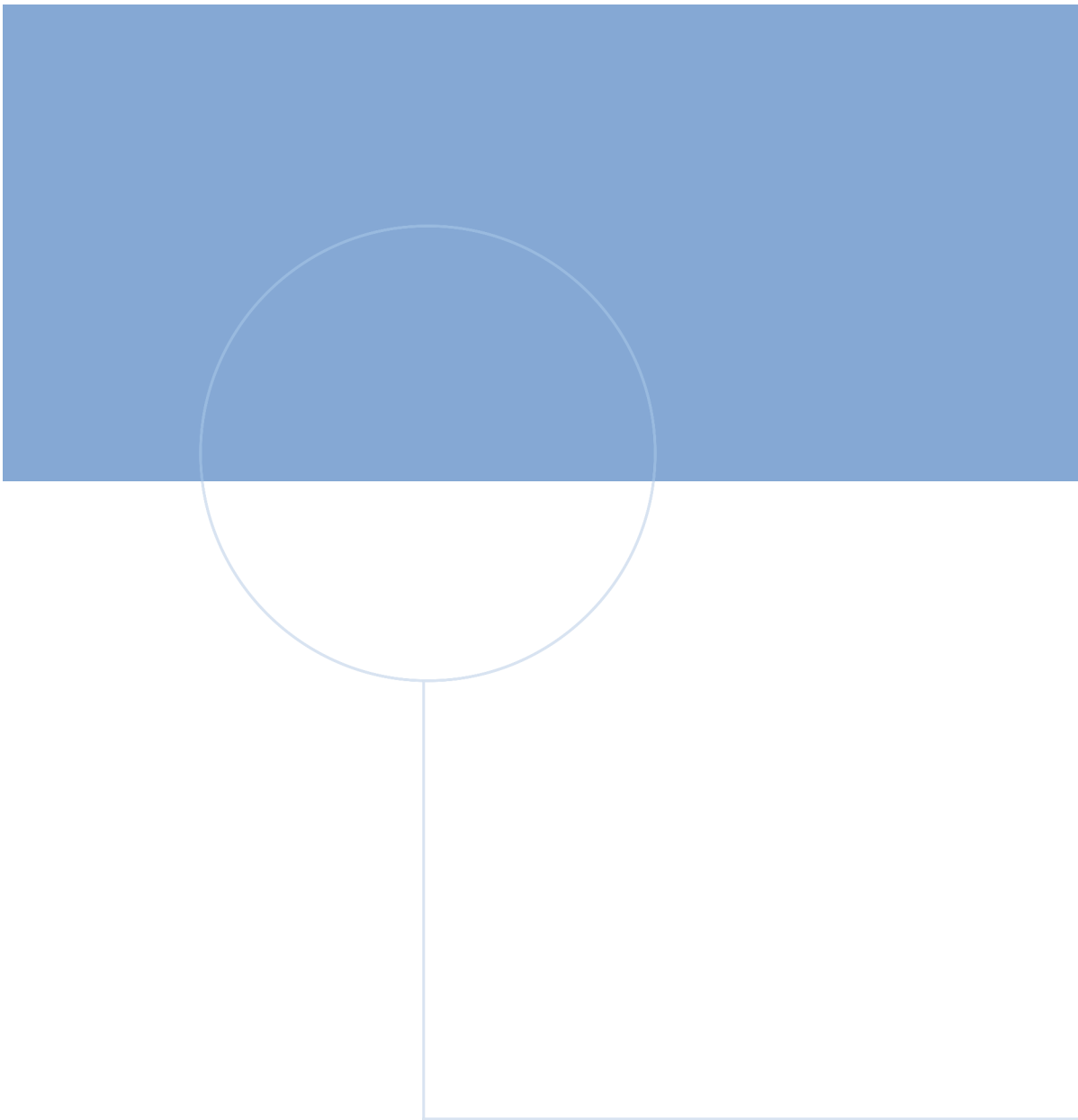
```
        ]
        y_test = y_test_total[
            test_machines_total.isin(
                machines[i * MachinesPerTrain : (i + 1) * MachinesPerTrain]
            )
        ]

        model.set_weights(parameters)
        loss, accuracy, precision, recall = model.evaluate(x_test, y_test)
        length = len(x_test)

        if i == 0:
            new_pred = model.predict(x_test)
            predictions = np.vstack((predictions, new_pred))

        i += 1
        try:
            x_test = x_test_total[
                test_machines_total.isin(
                    machines[i * MachinesPerTrain : (i + 1) * MachinesPerTrain]
                )
            ]
            new_pred = model.predict(x_test)
            predictions = np.vstack((predictions, new_pred))
        except:
            x_test = x_test_total[
                test_machines_total.isin(
                    machines[
                        (i - 1) * MachinesPerTrain : (i + 1 - 1) * MachinesPerTrain
                    ]
                )
            ]
            y_test = y_test_total[
                test_machines_total.isin(
                    machines[
                        (i - 1) * MachinesPerTrain : (i + 1 - 1) * MachinesPerTrain
                    ]
```

```
                )
            ]
            pass

        return loss, length, {"accuracy": accuracy}


if os.path.isfile("params.json"):
    with open("params.json") as target:
        params = json.load(target)


for param in params: #for every parameters in params.json file
    for repetition in range(repetitions): #repeat for repetition number of times
        MachinesPerTrain = int(25 / rounds)
        b1 = 50
        b0 = 15

        layers = [4, 16, 3]
        l_rate = 0.002
        model_params = []
        for item in param:
            if item != 3:
                model_params.append(tf.keras.layers.Dense(item, activation="relu"))
            else:
                model_params.append(tf.keras.layers.Dense(item, activation="softmax"))

        model = tf.keras.Sequential(model_params)

        model.compile(
            loss=tf.keras.losses.binary_crossentropy,
            optimizer=tf.keras.optimizers.Adam(learning_rate=l_rate),
            metrics=[
                tf.keras.metrics.BinaryAccuracy(name="accuracy"),
                tf.keras.metrics.Precision(name="precision"),
                tf.keras.metrics.Recall(name="recall"),
            ],
```

```
)

Ytag = ["Cycles"]
avg_stdev_file = "results/avg_stdev.csv"
train_file = "results/client"
test_file = "results/client"

(
    x_train_total,
    y_train_total,
    mu,
    sigma,
    Ymax,
    train_machines_total,
    originalY,
) = getData(train_file, 999)

machines = list(set(train_machines_total))

x_train = x_train_total[train_machines_total == machines[0]]
y_train = y_train_total[train_machines_total == machines[0]]
model.fit(x_train, y_train, epochs=100, verbose=0)

x_train = x_train_total[train_machines_total == machines[1]]
y_train = y_train_total[train_machines_total == machines[1]]
i = 0

predictions = [[None, None, None]]

x_test_total, y_test_total, test_machines_total, testY = getData(
    test_file, 999, mu, sigma, Ymax
)

y_train_total = y_train_total
y_test_total = y_test_total

# Start Flower client
```

```
try:
    fl.client.start_numpy_client(
        server_address="127.0.0.1:8080", client=Client()
    )
except Exception as e:
    print(e)
predictions = np.delete(predictions, 0, 0)
predictions = predictions.astype(float).round(0)

ts = datetime.now().strftime("%Y%m%d%H%M%S")

predictions = pd.DataFrame(predictions, columns=["Danger", "Alert", "Healthy"]
correct_predictions = y_train_total * predictions
if len(correct_predictions["Danger"].value_counts()) > 0:
    good_danger = (
        correct_predictions["Danger"].value_counts()[1]
        / predictions["Danger"].value_counts()[1]
    )
else:
    good_danger = 0

if len(correct_predictions["Healthy"].value_counts()) > 0:
    good_healthy = (
        correct_predictions["Healthy"].value_counts()[1]
        / predictions["Healthy"].value_counts()[1]
    )
else:
    good_healthy = 0

if len(correct_predictions["Alert"].value_counts()) > 0:
    good_alert = (
        correct_predictions["Alert"].value_counts()[1]
        / predictions["Alert"].value_counts()[1]
    )
else:
    good_alert = 0
print(
```

```
        "Good␣Predictions:\n--------------",
        "\nHealthy:␣",
        good_healthy,
        "\nAlert:␣",
        good_alert,
        "\nDanger:␣",
        good_danger,
        "\nParam:␣",
        param,
        "\nRepetition:␣",
        repetition + 1,
        "␣of␣",
        repetitions,
    )
    output = predictions.copy()
    output["Machine"] = test_machines_total
    output["Cycles"] = testY
    output["Correct␣Danger"] = correct_predictions["Danger"].copy()
    output["Correct␣Alert"] = correct_predictions["Alert"].copy()
    output["Correct␣Healthy"] = correct_predictions["Healthy"].copy()
    output.to_csv("output" + str(client) + ".csv", index=False)
    time.sleep(15)
```



NTNU

Norwegian University of
Science and Technology