UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF MECHANICAL ENGINEERING

# NEURAL NETWORKS FOR INTELLIGENT FAULT DIAGNOSIS OF MECHANICAL EQUIPMENT

by

**ILIAS KONSTANTINIDIS**

Submitted in partial fulfillment of the requirements for the degree of Diploma in Mechanical Engineering at the University of Thessaly

Volos, 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF MECHANICAL ENGINEERING

# NEURAL NETWORKS FOR INTELLIGENT FAULT DIAGNOSIS OF MECHANICAL EQUIPMENT

by

**ILIAS KONSTANTINIDIS**

Submitted in partial fulfillment of the requirements for the degree of Diploma in Mechanical Engineering at the University of Thessaly

Volos, 2021

## Approved by the Committee on Final Examination:

Advisor Dr. Ampountolas Konstantinos,
Associate Professor, Department of Mechanical Engineering,
University of Thessaly

Member Dr. Spiridon Karamanos,
Professor, Department of Mechanical Engineering, University of Thessaly

Member Dr. Grigorios Haidemenopoulos,
Professor, Department of Mechanical Engineering, University of Thessaly

Date Approved: September 29, 2021

## Acknowledgments

Writing this thesis marks the near end of five exciting academic years in the Mechanical Engineering Department of the University of Thessaly. Five years with a lot of challenges that hopefully prepared a new generation of young Engineers. On this small section I wholeheartedly want to thank my immediate family and friends, that supported me through the lows and cheered with me through the highs. I need to thank my parents for their remarkable emotional and financial support in those 5 years. I want to thank Dr Ampountolas for supervising this thesis, and who gave me the opportunity to study in more detail a subject that already intrigued me. Finally, I need to thank Dr Karamanos and Dr Haidemenopoulos for taking the time to read and evaluate my efforts.

# NEURAL NETWORKS FOR INTELLIGENT FAULT DIAGNOSIS OF MECHANICAL EQUIPMENT

ILIAS KONSTANTINIDIS

Department of Mechanical Engineering, University of Thessaly

Supervisor: Dr Konstantinos Ampountolas

Associate Professor University of Thessaly

## Abstract

The ever-growing scale and complexity of mechanical systems used on plant floors and research labs across the world, ordains the deployment of fast, reliable and - in the spirit of the 4th industrial revolution - fully automated solutions for the detection of possibly dangerous defects, resulted by their operation. Intelligent fault diagnosis is presented as a solution to the problem, and it relies on the usage of classification algorithms which are responsible to differentiate between the healthy state of the monitored mechanical system and the faulty state. In the context of the present work, different Artificial Neural Networks architectures, supervised learning methods and traditional classification techniques will be reviewed. After the theoretical ground for the methods mentioned above is laid, we will proceed to try them out on a case study in order to juxtapose their results and review each method's weaknesses and advantages. For the case study the rolling bearings dataset provided by the Case Western Reserve University (CWRU) will be used to detect faulty and healthy states of rolling bearings using vibrational data acquired from the CWRU's testing rigs.

## CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# Chapter 1. Introduction

## 1.1 Fault Diagnosis of Mechanical Equipment

When discussing fault diagnosis of mechanical equipment on the context of this work, we will be referring to the process of vibrational signal analysis, usually acquired from accelerometers attached to a rotating mechanical system being monitored (e.g., rolling bearing, turbines, fans), in order to classify the component as healthy or not. Vibrational signatures of rotating mechanical components are the most reliable fault indication, and they were used for traditional fault detection way before intelligent solutions like Neural Networks (NN) or other supervised learning methods were established. Fault diagnosis is quite a broad field of study, with varying implementations so the above lens is defined to help focus the context of the subjects being discussed.

## 1.2 Motivation and importance

Failures of rotating mechanical parts is something that needs to be expected when studying any mechanical system. Due to fatigue phenomena, every component of a larger system has the potential to fail, which can compromise the entire system and result in considerable productivity and economic losses. This realization drives the effort for robust and fast fault detection technologies which have the potential to provide warnings when a defected item is detected, so it can be replaced to ensure the safety and effectiveness of the overall system.

## 1.3 Literature review

Fault detection of mechanical equipment is a subject which interests the field's researchers since the mid-20th century, establishing several - now considered - traditional techniques. Piety, Magette [1] presented in 1979 a statistical technique for automatic fault detection, based on time and frequency domain descriptors to compose an overall signature that characterizes the health status of the component. R.B Randall [2] proposed in 1978 a methodology which utilizes Fast Fourier Transform (FFT) of the vibrational signal collected by faulty and healthy rotating parts to compare their frequency domain representations and classify the results. These traditional techniques have been the object of further research even in recent years, with papers verifying or improving on existing results [3] [4] [5].

Although traditional fault diagnosis methodologies have been proven to be effective, they focus their efforts on extracting the more obvious features of the vibrational signatures and as a result their performance is limited [6]. To combat that, research efforts are steered towards intelligent classification methods, such as NN. Eren et. Al. [7] proposed a Convolutional Neural Network (CNN) architecture for rolling bearing fault classification which exhibited great performance. Yuanhong Chang et. Al. [8] presented a different CNN architecture for detection of wind turbines faults. QiaoHu et. Al. [9] combined WPT with a supervised learning model called Support Vector Machine (SVM), while Diego Fernández-Francos et. Al. [10] used a one class SVM to differentiate between healthy and faulty conditions. SVM structures was also used by Junyan Yang et. Al. [11] on their work.

## 1.4 Thesis organization

The remaining of the thesis will be compartmentalized in four remaining chapters. First, several traditional fault detection techniques will be presented, alongside their respective historical context and their theoretical backbone. In continuum, the concepts of supervised machine learning algorithms will be introduced on a general mathematical and theoretical framework and then they will be recontextualized on the specific problem of fault classification of mechanical equipment. To close the third chapter, the relevant literature will be reviewed more closely, and specific structures and architectures of supervised methods will be thoroughly presented to lay the ground for the final chapter. Finally, the methodologies that have been introduced though out this work will be implemented on the problem of fault classification of rolling bearing defects, using a dataset from the public repository of CWRU. Traditional techniques and several intelligent methods will be applied, and their results will be documented and compared to help us draw our results.

# Chapter 2. Traditional Fault Diagnosis of Rotating Equipment

## 2.1 Introduction to Envelope Spectrum Fault Detection

The most used traditional fault diagnosis technique for rotating machinery in the relevant literature is Envelope Spectrum Fault Detection (ESFD). There are three general stages to follow in order to successfully diagnose a defected part using ESFD. First comes the data acquisition, followed by the signal processing which leads to the last stage and the goal, the diagnosis [12]. Mechanical components that comprise most of the literary work on the subject, are rotating equipment such as rolling bearings and gears.  This 3-step procedure mostly focuses on analyzing vibration signatures in form of time-series data acquired by accelerometers attached to the monitored system. Vibration signatures are demonstrated to be reliable indications of faulty equipment due to the way that a fault in a component influences its vibrational behavior [13].

## 2.2 Data acquisition

As it has been established by now, in order to perform health status diagnosis, vibrational signals need to be acquired by the part being monitored. To do that, some kind of vibration sensor needs to be attached to the part, with the most common types of sensors being accelerometers, tachometers, strain gauges and capacitive displacement sensors. For the purposes of this work, we will focus on accelerometers since they are the easiest and cheapest option to implement, and they are by far the most used type of sensor (Image 1.).

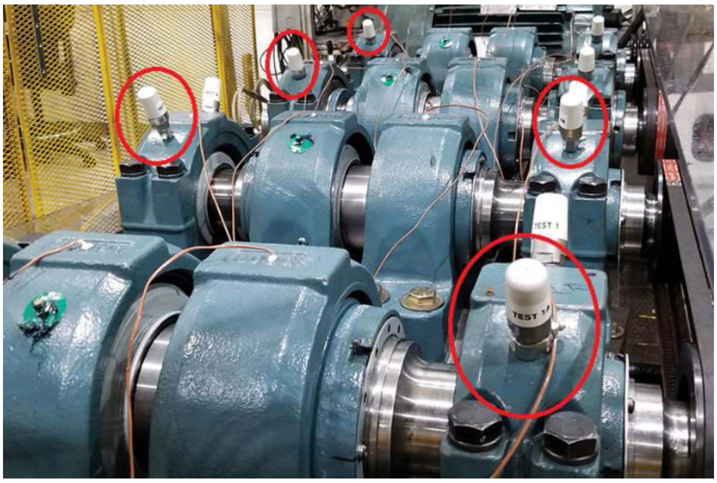


Image 1. [A] In this image four rotating shafts are presented, alongside their respective rolling bearings in both of their ends. In red, the vibration sensors (accelerometers) are highlighted, mounted on each bearing.

The typical micro electrical mechanical accelerometer (MEMA) is nothing more than a mass mounted on a spring, with some fixed plates as reference points. When the kinetic state of the accelerometer is altered due to an external force, the mass is displaced from its fixed-as-zero position and as a result the spring is moving along with the mass. Then the displacement of the spring is transformed into electrical signal, proportional to said displacement. [14]

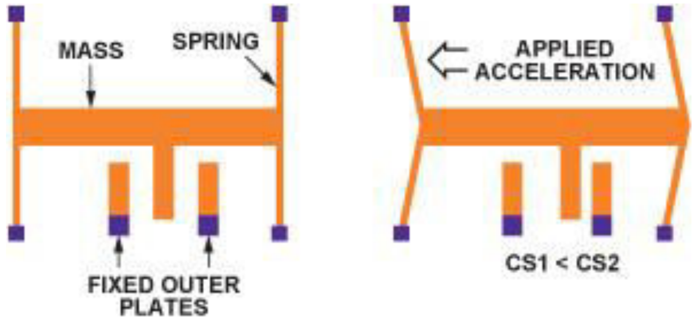


Image 2. [14] Schematic representation of a MEM accelerometer.

In order to acquire reliable and behaviorally descriptive data from the monitored component, as large a sampling rate as possible needs to be achieved. For example, the CWRU vibration signals dataset, was acquired using sampling rates of up to 48 kHz.  To achieve that, first and foremost an accelerometer with high enough sampling capability needs to be selected. A fast accelerometer however is able to measure as fast as the computational device that is connected to is able to record. To optimize the system, a dedicated computational device called Data Logger is connected to the sensor and is responsible for recording the electrical signals and move them to the last component of the system, the computer. There, the signal received from the accelerometer, through the data logger, is transformed to acceleration units and is ready for further analysis. The combination of all the components described above, comprises the Vibrational Data Acquisition System (VDAS) which is responsible for the first of our 3-stage procedure described previously [15].

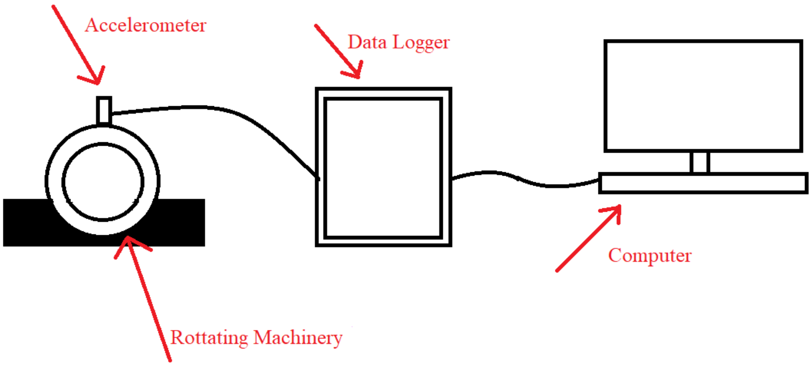


Image 3. VDAS system representation.

## 2.3 Signal processing

After the first step of the process is complete, and the vibrational data have reached the computer, signal analysis can take place. The data are recorded as discrete acceleration values on each sampling point, so they are represented in the time domain. In order to represent them in the frequency domain, the data need to be transformed using Discrete Fourier Transform (DFT):

$$X(k) = \sum_{n=0}^{N-1} x(n) * e^{-i2\pi kn/N} \quad [2.1]$$

Where, *N* is all the sample points, *n* is the respected sampling point, *k* discrete frequency *k = 0, 1, 2, …, N-1*

DFT analyzes the sequential discrete data into components of different frequencies, which can be very useful since it reveals periodicity in its input data. In order to perform the DFT efficiently and fast, the algorithm known as Fast Fourier Transform (FFT) is employed [16]. Also, in order to combat unwanted noise in our data, or intrusions of high energy nearby rotating machinery (or other environmental factors) an envelope filter needs to be applied on the time domain data, before the FFT. In order to remove the unwanted frequencies, the envelop filter imposes a bandpass filter around the band area we are interested in (more on that later) [17].

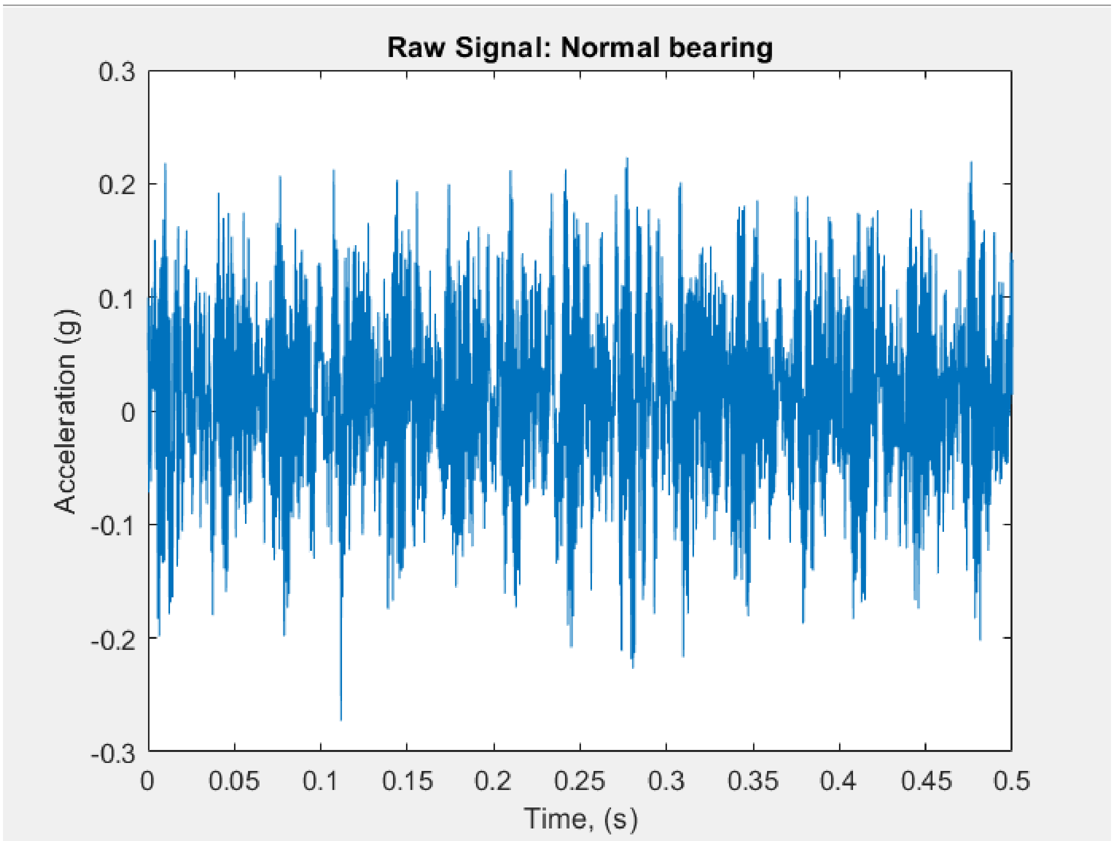


Figure 1. Time domain representation of a normal bearing, operating under 1772 rpm (CWRU Data).

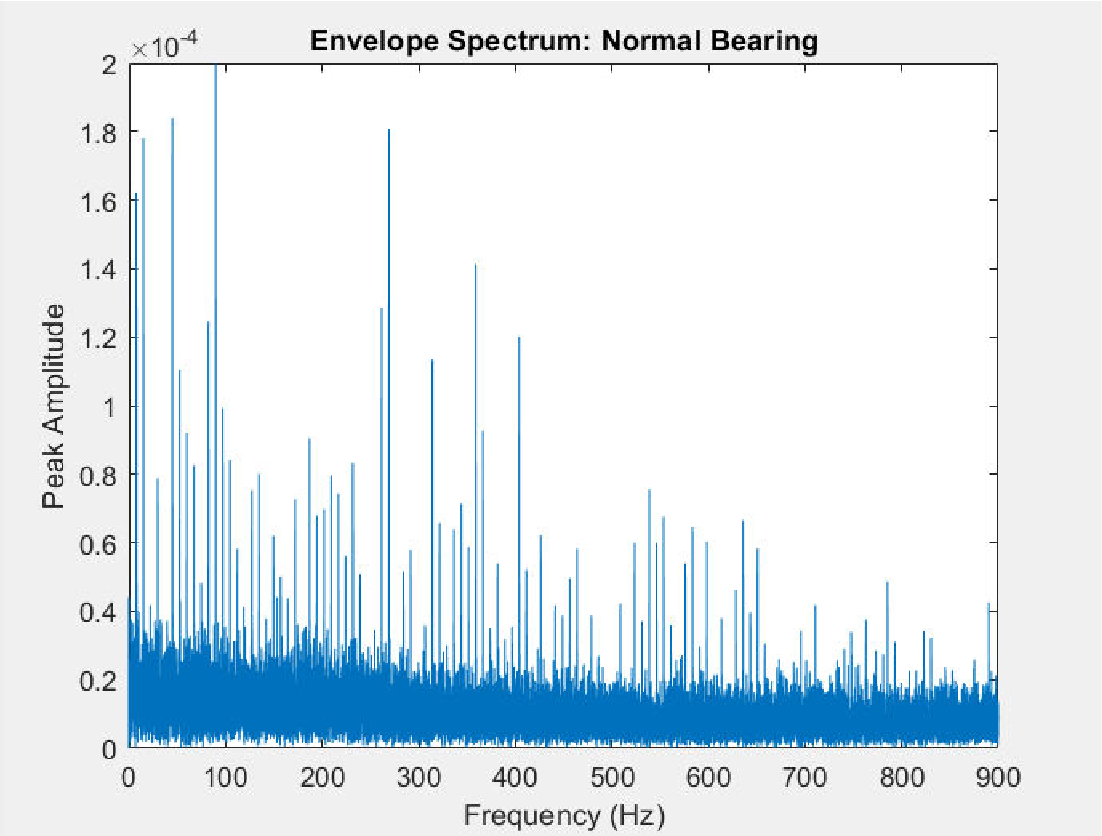


Figure 2. Frequency domain representation of a normal bearing operating at 1772 rpm (CWRU Data).

When a local fault exists in a rotating mechanical component, it produces an impact signal whose periodic nature can be accurately described by a specific frequency called fault characteristic frequency. One big shortcoming of this method is that these frequencies cannot be known a priori and in order to define them, a defected item needs to be isolated, put-on test rig and analyzed. One the other hand, the work on this field is quite extensive and as a result, empirical relations for most of the known faults of popular mechanical components such as gears, shafts and rolling bearings have been established [12].

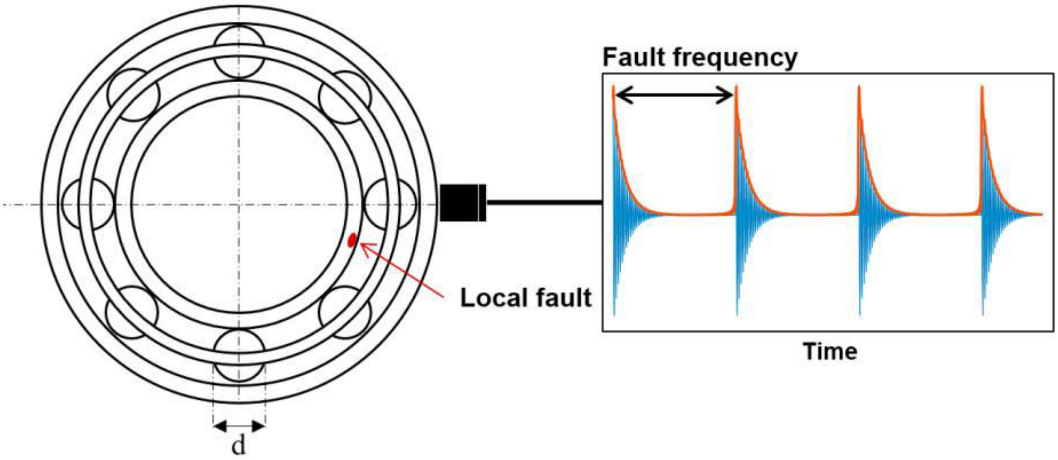


Image 4. [2] Local fault on rolling bearing and its periodic time-domain representation.

One very crucial aspect of the signal processing is the careful selection of the right band limits when applying the denoising procedures. A wider than it should bandwidth will hide the fault characteristic frequencies inside the noise, but a narrow one will block them from ever appearing on the frequency domain representation. There are two popularly used methodologies to follow when tackling the envelop filtering problem: squaring and low pass filtering the signal or using Hilbert transformation to analyze the signal.

● Following the first method, these next steps need to be implemented [12] [17]:

**A1)** Square the input signal which results in half the signal to be pushed to higher frequencies while the other half is shifted downwards.

**A2)** Down sample the signal in order to reduce the sampling frequency.

**A3)** Apply a minimum phase, lowpass filter to expunge the high energy frequencies.

**A4)** Amplify the signal by a factor of two, since only the low half of the original signal is preserved.

**A5)** Take the square root of the signal to fix the distortion introduced by the first step.

● Mathematically the envelope $e(t)$ of a signal $x(t)$ according to the second method is defined by the following:

$$e(t) = \sqrt{x^2(t) + \widehat{x^2}(t)} \quad [2.2]$$

Where $\hat{x}(t)$ is the Hilbert transform of $x(t)$.

Following the second method, these next steps need to be implemented [18]:

**B1)** Perform Hilbert Transform (HT) on the signal. Mathematically defined, the HT of a signal $x(t)$ can be described be the following:

$$H(x)(t) = \frac{1}{\pi} PV \int_{-\infty}^{\infty} \frac{x(\tau)}{t - \tau} d\tau \quad [2.3]$$

Where, *PV* is the Cauchy Principal Value.

A more useful way to calculate the HT of a signal is to use a 32-point Parks-McClellan FIR filter.

**B2)** Multiply the resulted HT with $\sqrt{-1}$ and add it to the time-delayed original signal to from the analytical signal.

**B3)** Take the absolute value of the analytical signal.

**B4)** Down sample and impose a low pass filter.

Both can be very effective, but the HT method can better handle signals that are aliased due to great concentration of high frequencies.

## 2.4 Fault Diagnosis

As a result of the above, it is reasonable to expect that if a rotating mechanical component presents a local fault, its envelope spectrum will depict amplitude spikes on its fault frequency and on the harmonics of that frequency. That way it can be concluded visually, by observing the envelope spectrum of a monitored component, not only if it is faulty but also the location of the fault (provided that every possible fault frequency for every possible local failure is known) [12].

As an example, using the CWRU Dataset, two different envelope spectrums of faulty equipment are presented below. The example presents one rolling bearing with fault on its inner ring, and the second on its outer ring. The rolling bearings used in the example have identical specifications, their rotational speed is 1772 rpm, and their fault frequencies can be calculated in accordance with the table 1 provided by CWRU.

| Fault frequencies, multiple of running speeds in Hz. | |
|---|---|
| Inner ring | Outer ring |
| 5.4152 | 3.5848 |

Table 1. [III] Fault Frequency table (CWRU Data).

Therefore, the fault frequencies can be calculated to be 159,9287 Hz for the inner ring fault and 105,8710 Hz for the outer ring.

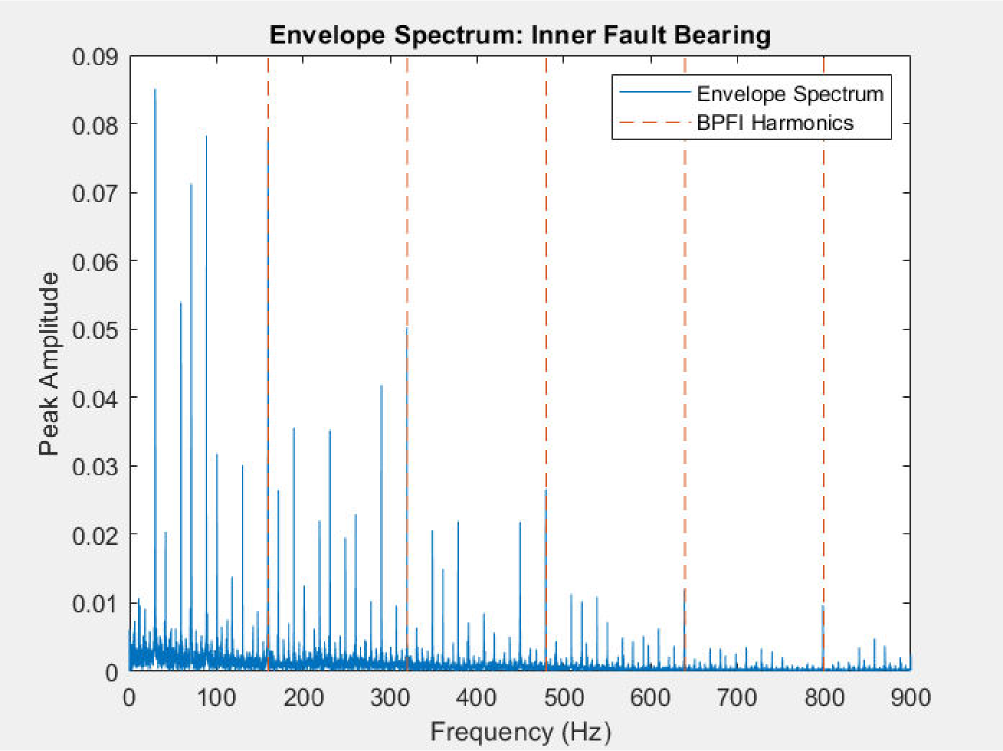


Figure 3. Envelope spectrum of a bearing with inner race fault operating at 1772 rpm (CWRU Data).

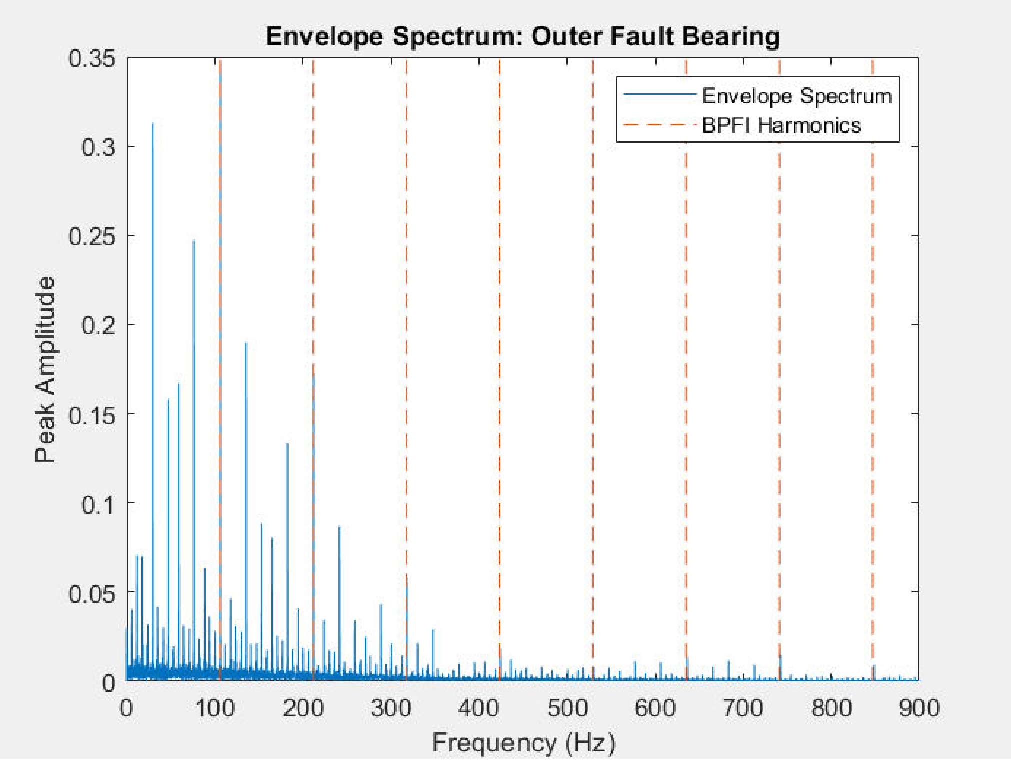


Figure 4. Envelope spectrum of a bearing with outer race fault operating at 1772 rpm (CWRU Data).

As it can be observed on the figures 3 and 4, the fault fundamental frequencies of the inner and outer faults coincide exactly with some of the highest peaks on their respective areas. Therefore, it is reasonable to conclude that the monitored bearings are indeed faulty.

On the other hand, if the same procedure is repeated, but this time the vibrational data will be derived by a healthy bearing, the following envelope spectrums will be produced (figures 5 and 6).

As it can be observed on the figures 5 and 6, the fault fundamental frequencies do not coincide with any amplitude peaks and are unable to demonstrate any kind recurring pattern in relation to the frequencies that they do coincide with. Therefore, it is reasonable to conclude that the monitored bearing does not demonstrate nor inner race fault, neither outer race fault.

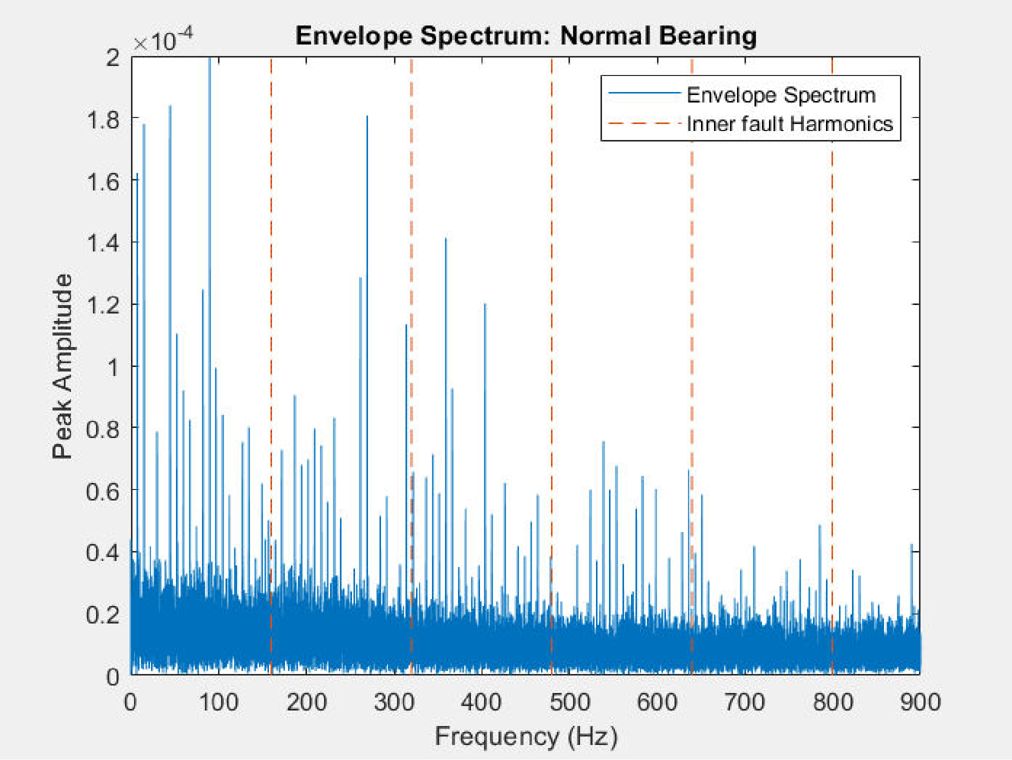


Figure 5. Envelope spectrum of a normal bearing operating at 1772 rpm (CWRU Data).

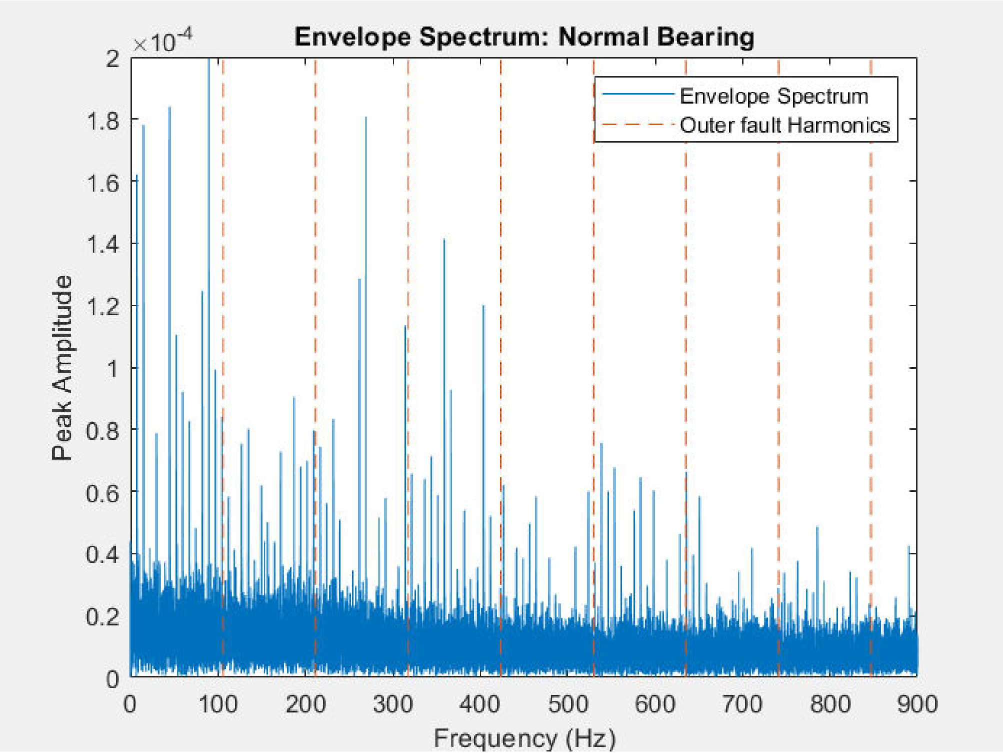


Figure 6. Envelope spectrum of a normal bearing operating at 1772 rpm (CWRU Data).

## 2.5 Other Traditional Fault Diagnosis Techniques and ESFD

In the literature concerning fault diagnosis of rotating machinery, several other techniques have been established. Temperature monitoring, acoustic emission analysis, wear debris analysis, nondestructive tests, statistical kurtosis analysis and shock pulse monitoring, just to name a few different approaches that can be found [19]. Despite the existence of all the alternatives mentioned above, EFSD has managed to prevail as the most used in both the relevant academic body of work in the subject, as well as in its usage in industrial environments. It can adapt to the needs of different applications, provided that fault specific details are known, in addition to its ability to provide the user with detailed information about the location of the fault. As a result of the above, EFSD was chosen as the focus of this chapter and as the traditional technique to be used as a comparison for the modern intelligent methods that will be presented in the remaining of the work.

## 2.6 Chapter Conclusions

In this chapter the most commonly used traditional fault diagnosis technique for rotating equipment has been presented. A combination of FFT and envelope filtering techniques have the potential to discard all the noise that is inherent in data acquisition and unsheathe the health condition of the monitored component. Unfortunately, this does not come without drawbacks, since in order to classify a rotating component as faulty or not, all the possible faults a component can present must be already known, as well as their specific fundamental frequencies. This reality introduces a lot of a priori knowledge that might not be available in any given possible application. Additionally, even if FFT is considered to be a very efficient way to perform Time-Frequency decomposition, is quite computationally expensive which can cause severe problems when trying to implement the method on a Real-Time environment.

Closing this chapter, the ground has been laid for the introduction of more intelligent ways of tackling the problem. Even if the concepts that will be discussed in continuum will be quite different, they are a built on the same 3-step procedure of Data Acquisition, Signal Processing and Diagnosis.

# Chapter 3. Intelligent Fault Diagnosis of Rotating Equipment

## 3.1 Introduction

Intelligent fault diagnosis maintains the same 3-step core that has already been presented in the previous chapter. Vibrational data needs to be acquired from the monitored system and then got through some processing to reach a health diagnosis. The way the processing step works though, is fundamentally different. In the place of domain transformations and fault specific frequency calculations, supervised learning classification algorithms are being employed. The biggest advantage of this approach is that these algorithms are able to classify the vibrational signal as heathy or faulty, based on close-to raw time series data, thus eliminating the need for time and resource consuming domain transformations or other preprocessing procedures.

The main tool intelligent fault diagnosis uses to classify the vibrational samples it receives are Neural Networks (NN). These kinds of algorithms are designed to mimic the way biological preceptors, like the human brain, are working in order to detect patterns and conjunctive relations in the data they are fed with. There are two large categories in which Neural Networks can be subdivided, supervised and unsupervised learning, but in the context on this work the focus will be on supervised learning methodologies. That term refers to machine learning algorithms which utilizes a weighted function called prediction function. The weights of this function are determined by a repeating exposure of the function on pairs of input and their desired output. This repeating procedure is referred to as training of the Neural Network. The goal is, provided that the data are descriptive enough of the problem in hand, to create a precise enough prediction function during the training, that it can predict the right result for any input that is being fed with. That way if a function is trained good enough on a big dataset of vibrational signal inputs of mechanical equipment and their corresponding health state (e.g., healthy, ball fault, etc.), the resulted function will be able to predict the health state of any given vibrational signal. Because we are directing the NN by providing it with the right answers in the training data, this procedure is called supervised learning.

## 3.2 Neural Networks: A Basic Framework

The fundamental concept of NN [20] can be mathematically expressed by the following simple relation.

$$y = \varphi(x) \quad [3.1]$$

Where $y$ is the output, $\varphi$ is the prediction function and $x$ is the input.

The main goal of the NN is to determine a function $\varphi$ which, given a value $x$, can relieably predict the value $y$. A few other basic concepts need to be understood to frame the above relation in a clearer context, are the following:

● Raw data: The data just as they have been collected by the sensors. A sensor could be an accelerometer, a camera, a pressure sensor, it depends on the context of the application.

●Labels: A value that is attached to every single one of the raw data samples and describes the class this data sample belongs to.

● Dataset: The data structure that contains both the collected data samples (raw data is the desired approach, but it is not always feasible) and their corresponding labels.

● Preprocessing: Data rarely come in neat formats, ready to be imported in the NN, so some light formatting in appropriately sized and dimensioned matrixes usually should take place. It is also common to do some data mapping, like normalization before the next stages.

● Training Procedure: As already stated, the repeated exposure of the NN to a part of the Dataset (called Training Dataset), which aims to determine the prediction function $\varphi$. Training is a fundamentally feedforward procedure.

● Testing Procedure: The exposure of the NN to a different part of the Dataset (called Testing Dataset) with which it had no previous interaction, but this time the NN only "sees" the input, tries to predict the output using the $\varphi$ function and then cross validates the label it produced with the real label. This process also defines and calculates a performance metric (e.g., accuracy, precision, etc.) which is needed to decide whether the training was successful, and whether the prediction function has gained the ability to actually predict the label of any given data sample it encounters for the first time.

● Validation Dataset: A part of the dataset that its samples are not part of the training dataset, but they are used in the training procedure at the end of each training iteration to provide the Network with an unbiased evaluation of its performance on that particular iteration. What the model learned from the validation data though are incorporated into the model, which can lead to a more biased NN. Its usage is not necessary.

● Prediction: When all the procedures described above are completed, the resulted prediction function is ready to be fed with new data and classify them in accordance with its training.

The following flow chart represents the consecutive process of building a supervised learning model with the concepts that have been introduced.

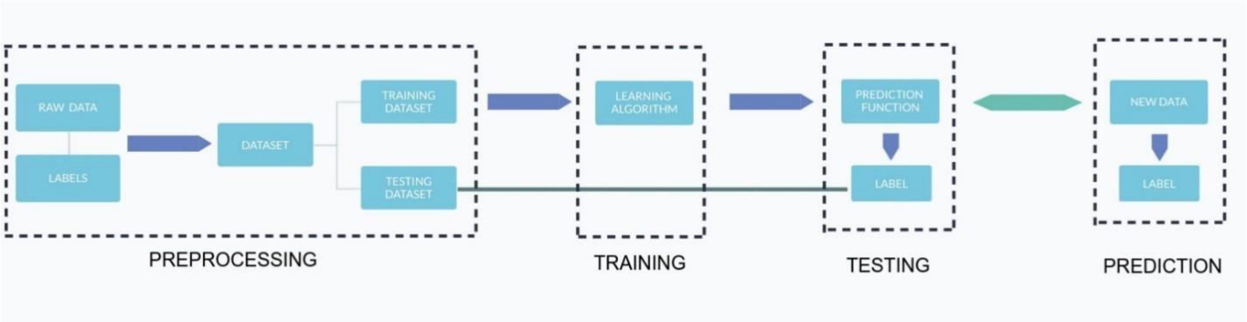


Image 5. Flow chart for machine learning model building.

As it has been established by now, the core of any machine learning problem is to determine a good enough prediction function $\varphi$. So, it stands to reason to focus on this, more complicated than it initially seems, problem. First and foremost, the equation 3.1 is not actually neither what $\varphi$ is expected nor what it is been demanded by it to do. A better mathematical representation of the NN goal when it tries to determine the prediction function is the following.

$$y_i \approx \varphi(x_i; a) \quad for\ i = 1,2, \dots, m \quad [3.2]$$

Where, $y$ is a vector containing class labels, $x$ is a vector containing the input data and $m$ is the population of the pairs data samples – class labels.

The prediction function can be defined in terms of some parametric vector $a$, thus the problem can now be framed as a data fitting optimization problem, with an objective of finding the best $a$ to map every $x_i$ to a value $y_i$. To calculate this vector $a$, the NN selects through its training the parameters $(W^A, g^A),\ A = 1,2,3, \dots, J$ . These parameters are usually called weights $W$ and biases $g$. The training of the NN, essentially is the calculation of this parameters which ultimately define the prediction function φ, through successive transformations to the input vector $x_i \in \mathbb{R}^{d_j}$. These transformations are made in *layers*, the type of layer defines the type of the transformation that takes place. The first layer receives as input the $x_i$ vector and outputs a vector that represents the odds of the input to belong in each of the classes that the problem is working with. In continuum, the output is being fed on the next layer and so on until all the defined layers and their corresponding transformations have taken place. An example of a layer is a canonical fully connected layer, which performs the following element-wise transformation [21].

$$x_i^{(j)} = \sigma\left(W^j x_i^{(j-1)} + g^j\right) \quad [3.3]$$

Where $x_i^{(0)} = x_i$, $W^j \in \mathbb{R}^{d_j \times d_{j-1}}$ and $g^j \in \mathbb{R}^{d_j}$ contain the $jth$ layer's weights and biases, $\sigma$ represents the element-wise activation function which is an inseparable part of any layer and is responsible for the way that the weighted sum of the input is being moved to the next layer. Every layer contains a number of nodes which are the actual part of the network that the transformations take place, their number and size is determined by the designer but is relevant to different things like the size of the input, the kind of layer they comprise and more.

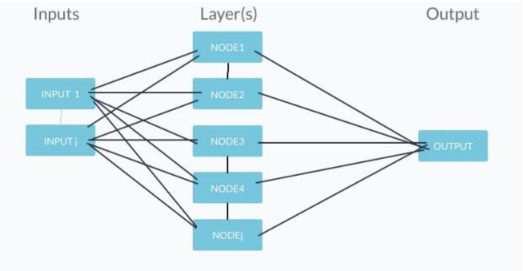


Image 6. Schematic representation of the structure of simple one-layer-deep Neural Network.

Generally speaking, there is no particular rule of thumb to follow when selecting the number and kind of layers, the number of nodes, their size, the loss function or the activation function. There are a few empirical observations of approaches that work best, but they are by no means absolute rules and the safest approach to select the proper architecture for the task in hand, is a combination of experience with the subject matter and performing comparative tests of different architectures.

Through this successive procedure, the final output vector $x_i^{(J)}$ results to the calculation of a prediction function $\varphi(x_i; a)$, where the parameter vector $a$ is the collection of weights and biases acquired by each layer $\{(W^1, g^1), \dots, (W^J, g^J)\}$. Of course this has to repeat for every sample of the dataset.

As a logical consequence of the above, the following objective function needs to be minimized in order to find the $a$ vector which better defines the prediction function $\varphi$.

$$\sum_{i=1}^{m} h((x_i; a), y_i) \quad [3.4]$$

Where $a = (W^1, g^1, W^2, g^2, \dots)$ which are the weights and biases the NN calculates and $h$ measures the mismatch between the real $y_i$ and the $y_i$ produced by the model [20].

As a result, the goal of the training of a NN can be redefined as the minimization of the difference between the real samples and the samples produced by the NN.

In order to frame this mismatch that the function $\varphi$ represents in a manner that can actually be used to perform the optimization, it can be expressed as the *expected loss* $L(\alpha)$ . In a perfect world, the parametric vector $\alpha$ is minimizing the expected loss for any input-output pair. This would entail for one, to know a probability distribution $P(x, y)$ which is able to encapsulate the true relationship between the dataset pairs. Assuming that the input-output space $\mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$ is endowed with $P: \mathbb{R}^{d_x} \times \mathbb{R}^{d_y} \rightarrow [0,1]$ , then the objective function that needs to be minimized i.e., the expected loss function, is defined as follows.

$$L(\alpha) = \int_{\mathbb{R}^{d_x} \times \mathbb{R}^{d_y}} loss(\varphi(x; a), y)\, dP(x, y) = \boldsymbol{E}[loss(\varphi(x; a), y)] \quad [3.5]$$

Where $\boldsymbol{E}$ is the expected value of $\varphi(x)$ and $loss$ is a function which provides a continuous approximation of a cost measurement for predicting the value of $\varphi$ when actual label is $y$.

Unfortunately, to calculate the $E$ a possibility function $P(x, y)$ is needed and there is no way to estimate it. So, in the place of expected loss, the *empirical loss* $L_e$ can be used. Mathematically, empirical loss can be defined in $L_e: \mathbb{R}^d \rightarrow \mathbb{R}$ as follows.

$$L_e(\alpha) = \frac{1}{m}\sum_{i}^{m} loss(\varphi(x_i; a), y_i)) \quad [3.6]$$

Where $m \in \mathrm{N}$ and represents the size of a dataset $\{(x_i, y_i)\}_{i=1}^{n} \subseteq \mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$.

Finally, to redefine the problem one last time, the goal of the training of a NN is the minimization of the empirical loss $L_e$ [21].

This optimization problem though usually is highly nonlinear and nonconvex, which makes it near impossible to solve in a global minimum. To counter this, gradient methods have been used in the relevant literature, resulting in sufficiently good approximations. The fundamental observation these gradient based approaches use, is that *the gradient of the objective function [3.6] with respect to the parametric vector $a$ can be computed by the chain rule, using algorithmic differentiation [21].* Machine learning researchers refer to this technique as *backwards propagation*, and it is the bedrock of modern Neural Networks. Backpropagation, for short, is responsible for computing the gradient of the empirical loss function in respect to its weights and biases i.e., the vector $a$, but how the gradient is used to optimize the objective is bound by the optimization algorithm that one chooses to apply on the NN. Famous examples of optimization algorithms are *stochastic gradient (SG)*, *batch gradient descent,* etc.

It is important to note that there is a good reason that equations 3.1 and 3.2 have different comparative operators ($=$ and $\approx$, respectively), and that the discussion revolves around approximate minimization of the empirical loss. Even if the optimization procedure could construct an exact minimizer of the stated objective function (like the $=$ implies in equation 3.1) that is decidedly not desired, because it would constitute overfitting the prediction function $\varphi$. An overfitted $\varphi$ will be excellent in predicting the behavior of the data that is trained on, but it becomes so tailored to them that is unable to unsheathe the features on any new data it encounters. Overfitting usually is the result of prolonged training or insufficient volumes of data, and it can be countered by setting an upper threshold on the desired metric which will terminate the training when reached [20]. For example, if an accuracy threshold of 95% for the predictions is set, the training will end if reached before the predetermined time. Also using less data from the set, implementing regulation and constructing as simple a NN as possible can protect the model from overfitting on its training data. All the above, work in service of a very important property that the final solution is expected to have, generalizability. It is essential for a NN to be able to provide its user with reliable label classification on as big a range of data as possible. [20]

## 3.3 Optimization methods for Neural Networks

Since the basic lens through machine learning is viewed i.e., the minimization of empirical loss, has been established, the next step is to review exactly how this optimization problem can be resolved. To do that, the concept of *gradient descent* needs to be formally introduced. On top of this pretty straight forward concept, the majority of the most used optimization algorithms for NNs are built. It is also important to point out that for the remaining of the chapter 3.3, the assumption is made that the minimized functions are continuously differentiable on $\mathbb{R}^n \to \mathbb{R}$ , that full gradients can be computed in each iteration but also that the functions are not necessarily convex.

### 3.3.1 Gradient Descent

Assuming a hypothetical multi-variable function $F$ is differentiable on point $x$, assuming $x$ is part of the domain of $F$, then $F$ decreases the fastest in the direction of the negative gradient of $F$ on $x$ i.e., $-\nabla F(x)$. The logical continuum of this observation is that if $x_{n+1}$ are also part of $F$'s domain and defined like so:

$$x_{n+1} = x_n - \gamma \nabla F(x_n) \quad [3.7]$$

with $\gamma \in \mathbb{R}_+$ small enough for the following to be true.

$$F(x_n) \ \geq F(x_{n+1}) \quad [3.8]$$

To contextualize the above, one can take small steps (the size of the steps is defined be $\gamma$) by subtracting the term $\gamma \nabla F(x_n)$ from $x_n$, while fiddling with the size of the step $\gamma$, until a local minimum for $F$ is reached. In theory, in the $(n+1)$ in which the relation [3.8] is no longer true, the value of $x_n$ is the local minimum. If the hypothetical function $F$ is convex, then all local minima are also global, so an absolute minimization can be achieved.

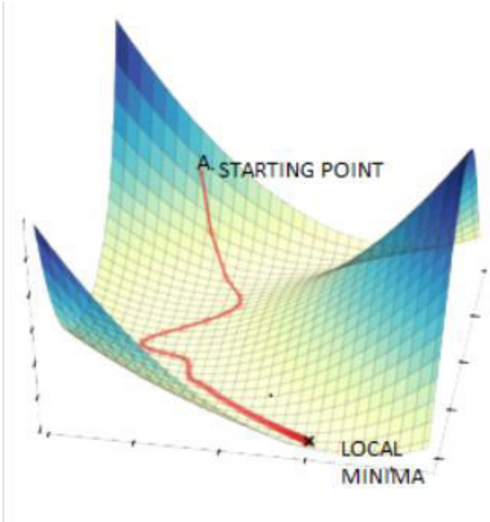


Image 7. [B] Visual representation of gradient descent for a 2-variable function.

One very important aspect of the algorithm presented above, is how one goes about choosing a $\gamma$ to use in each step. One approach would be *exact line search*, which decides $\gamma$ using the following relation.

$$\gamma_{n+1} = \ argmin_n[x_n - \gamma_n \nabla F(x_n)] \quad [3.9]$$

This method is reliable, but quite computationally expensive, so another method that is being derived from exact line search and constitutes the most popular way of selecting $\gamma$ for every iteration of the gradient descent algorithm is *backtracking line search.* In this method, parameters $c \in \left(0, \frac{1}{2}\right)$ and $b \in (0,1)$ are assumed and then the multiplication $x_{n+1} = bx_n$ is performed until the following condition is satisfied. Then, the inequality becomes an equality, and it is resolved for $\gamma_n$.

$$F(\ x_{n+1} - \gamma_{n+1} \nabla F(x_{n+1})) \leq F(x_n) - \ c\gamma_n \parallel \nabla F(x_n) \parallel^2 \quad [3.10]$$

The parameter γ is also known as *learning rate* in the context of machine learning optimization [20].

Gradient descent is a very elementary but effective optimization algorithm which entails some computationally expensive steps, especially for large datasets like the ones that are being used for machine learning. As a direct result of this realization, some of the most used and useful optimization schemes for machine learning applications are based on it but are cleverly tweaking its formula to make it even more effective and faster. Some of these methods will be presented in the remaining of this subsection are full gradient, accelerated gradient, stochastic gradient, ADAM, etc. Although, it should be noted that in some cases using gradient descent is still doable and actually quite effective, for example shallowly trained NN demonstrate sufficient results even with the most elementary gradient descent implementation.

It is quite noteworthy that some of the most useful optimization techniques are not modern approaches, but they were actually conceived as far back as 1951, when Robbins & Monroe published their, now classic of machine learning literature, paper:

A stochastic gradient approximation [22]. Other examples are Frank & Wolfe (1956) Conditional Gradient [23], Bertsekas and Tsitsiklis (1989) Parallel coordinate descent and incremental gradient algorithms [24] and Eckstein & Bertsekas (1991) Alternating direction method of multipliers (ADMM) [25]. It is interesting that the theoretical bedrock for artificial intelligence (AI) was laid so far back, but the computational complexity of these algorithms was too much for the computers of the time to cope with.

It is important to note that rarely, if ever, gradient descent methods converge in only one training session upon the training dataset. So, it is very important to perform multiple passes of the entire dataset through the network. The number of times the training upon the entire dataset is repeated is called an *epoch*.

### 3.3.2 Stochastic Gradient Descent

One of the most elementary but powerful improvements upon gradient descent is stochastic gradient descent (SGD) [22]. Commonly viewed as a stochastic approximation of gradient descent, SGD instead of computing the full gradient using the entirety of the dataset, it calculates an estimate derived by a randomly selected subset of the dataset. The goal of SGD is to minimize the empirical loss function with the smallest possible computational complexity, in expense of a lower convergence rate (meaning more time until the optimization finishes). Mathematically, SGD can be expressed as follows.

$$a_{n+1} \leftarrow \; a_n - \gamma\nabla\big[loss\big(\varphi(x_i; a_n), y_i\big)\big]_n \quad [3.11]$$

Where $n \in N$, $a = (W^1, g^1, W^2, g^2, \dots)$ as presented in subception 3.2, $a_1$ is given, the pair $(x_i, y_i)$ is randomly selected from the entirety of the dataset and $\gamma$ is a positive constant step also known as learning rate.

As a result of the above, each iteration of the algorithm calculates just the gradient $\nabla\big[loss\big(\varphi(x_i; a_n), y_i\big)\big]_n$ and nothing else before the parameters of the NN update, which leads to really computationally cheap iterations. These frequent updates also help the algorithm to move out of local minima due to the violent way they change the value of the learning rate in each iteration. Another feature that differentiates the method from the crowd is the non-deterministic aspect of its implementation. The iteration sequence is not determined only by fixed resources, namely the loss function $L_e$, the starting point $a_1$ and the step size $\gamma_n$, but also by the random selection of the data point $(x_i, y_i)$.

On the downside, the effect on the computational expense cannot be noticed unless the dataset is really large and in fact the intensity of the large number of iterations might harm the overall computational cost in smaller datasets. Also, the very feature that defines SGD, the fact that each step of the descent is computed from just one sample, is possible to have the opposite of the intended effect and steer the descent into completely wrong direction.

### 3.3.3 Batch Gradient Descent

Another really useful and frequent variation of gradient descent is Batch Gradient Descent (BGD). This optimization algorithm follows the formula of gradient descent much closer than SGD did and it does not use just one sample to derive the network parameters before each update. In contrast, BGD compartmentalizes the dataset in much smaller mini datasets called *batches.* Then, each batch is fed into the NN, their gradient is computed, and then the weights and biases are updated. Mathematically BGD can be expressed like so.

$$a_{n+1} \leftarrow a_n - \gamma_n \nabla L_{e_n}(a_n) \quad [3.12]$$

Where $n \in N$ and represents the number of batches, $a = (W^1, g^1, W^2, g^2, \dots)$ as presented in subsection 3.2, $\gamma_n$ is a positive constant step selected using backtracking line search as presented in subsection 3.3.1, or even be selected a steady value depending on the implementation and $L_e$ is the empirical loss function defined in 3.6.

Using BGD requires the computation of a gradient which entails the parameters derived from all the data points in a batch. This could be quite computationally expensive, compared to SGD, but the result is expected to be significantly better oriented towards the minimum of the function [20] [21].

Both SGD and BGD are two of the most elementary and effective optimization methodologies there are. As a result of this it stands to reason to frame in a comparative way, as they have already been presented. They also provide the basis for building much of the more complicated algorithms used in industry and academic level.

### 3.3.4 Gradient Descent with Momentum

Also commonly referred to as *heavy ball method*, Gradient Descent with Momentum (GDM) is yet another differentiation of the original gradient descent algorithm. On this alteration, each step of the descent is computed by a combination of the steepest step direction and the by the difference of the last two iterations. Mathematically this procedure can be expressed like so.

$$a_{n+1} \leftarrow a_n - w_n \nabla L_{e_n}(a_n) + b_n(a_n - a_{n-1}) \quad [3.13]$$

Where $n \in N$, $a = (W^1, g^1, W^2, g^2, \dots)$ as presented in subsection 3.2, $w_n$ and $b_n$ are scalar sequences that can be either set dynamically or be predetermined and $L_e$ is the empirical loss function defined in 3.6. The term $b_n(a_n - a_{n-1})$ is often referred to as *momentum* term and, depending on the intensity of $b_n$, regulates the direction of the descent and does not allow great oscillations from step to step.

There are two main approaches for deciding the values of $w_n$ and $b_n$. The first is setting a fixed value, for which $w_n = w > 0$ and $b_n = b > 0$. The optimal selection of this global values is the result of trial and error, but the more experience one has with this method the easier it gets to choose them effectively. There also analytical ways to find the optimal $w$ and $b$, but they required knowledge of information that might not be available, for example the minimum and maximum eigenvalues of the minimized function. Alternatively, if and only if the objective function is strictly convex quadratic, then $w_n$ and $b_n$ can be optimally selected for each $n \in \mathbb{R}$ by finding the pair that satisfies the following condition.

$$min_{(w,b)}\left[L_e\left(a_n - w\nabla L_e(a_n) + b(a_n - a_{n-1})\right)\right] \quad [3.14]$$

This method has demonstrated significantly superior results regarding the rate of convergence from a simple gradient descent, both with stationary and with dynamic parameter selection. The key difference between the two approaches, is that the dynamic method - even though it is computationally more expensive, and its convergence behavior is more complex than the linear convergence rate of the stationary method – provides finite convergence guarantee, while its stationary counterpart does not [20] [21].

As it is expected, GDM can work both with stochastic and batch approaches, inheriting each methods problems and advantages.

### 3.3.5 Accelerated Gradient Descent

Another similar, but distinctly different method is Accelerated Gradient Descent (AGD) or Forward-Backward Method, as its author named it [26]. It can be viewed as a reversal of the GDM method. If we consider the GDM procedure to act by taking the steepest descent step first and then applying the momentum term, the AGD acts in the opposite way by letting the effect of the momentum term take place first and from that point taking the steepest descent step. This procedure can be represented as a two-step approach like so.

$$\tilde{a}_n \leftarrow a_n + b_n(a_n - a_{n-1}) \quad [step\ 1]$$

[3.15]

$$a_{n+1} \leftarrow \tilde{a}_n - w_n \nabla L_{e_n}(\tilde{a}_n) \quad [step\ 2]$$

Where $n \in N$, $a = (W^1, g^1, W^2, g^2, \dots)$ as presented in subsection 3.2, $w_n$ and $b_n$ are scalar sequences that can be either set dynamically or be predetermined and $L_e$ is the empirical loss function defined in 3.6.

The distinction between AGD and GDM presented above could seem minor. On the contrary though, it has been demonstrated that for $w_n = w > 0$, for all $n \in N$, for $b_n \nearrow 1$, convex and continuously differentiable $L_e$ and Lipschitz continuous gradient the optimal complexity of iterations can be achieved. In fact, for the exact same case the GD method would converge in a rate of $\mathcal{O}(\frac{1}{n})$, while the AGD method would converge in a rate of $\mathcal{O}(\frac{1}{n^2})$. Obviously, the difference between the two is huge and this result is considered to be, in the time of writing, the fastest converging rate for a traditional gradient descent-based method there is [20] [21].

On the other hand, while the rate of descent for AGD is quite impressive, the computational burden to perform the iteration [3.15] is quite heavy. As a result of the above, caution is advised when AGD is implemented because the overall time until convergence could be larger than other methods due to the size of the dataset. As a rule of thumb, bigger sets tend to gain the most from computationally expensive methods like AGD.

### 3.3.6 The Adam Optimizer

The final optimization algorithm that will be presented on the context of this work, is the Adam optimizer. Interestingly enough, Adam is not an acronym, and the name is rooted in the term *adaptive moment estimation* which is the underlying concept the optimizer is based on. The algorithm is an advancement of the previously reviewed method SGD and has proven itself to be one of the most computationally efficient algorithms there is, even for quite noisy or sparse gradients. Since its conception in 2015, it has become one of the staples of machine learning due to its straightforward implementation, efficiency and universality with minimal tuning.

Adam is a prime example of modern approaches in machine learning optimization algorithms, so it is quite more complex than any of the algorithms previously presented in this work. It can be compartmentalized into six distinct steps that are responsible for each iterative descent step.

$$g_{n+1} \leftarrow \nabla\big[loss\big(\varphi(x_i; a_n), y_i\big)\big]_n \quad [step\ 1]$$

$$m_{n+1} \leftarrow b_1 m_n + (1 - b_1) g_{n+1} \quad [step\ 2]$$

$$u_{n+1} \leftarrow b_2 u_n + (1 - b_2) g_{n+1}^2 \quad [step\ 3]$$

$$\widehat{m}_{n+1} \leftarrow \frac{m_{n+1}}{1 - b_1^{n+1}} \quad [step\ 4]$$

$$\hat{u}_{n+1} \leftarrow \frac{u_{n+1}}{1 - b_2^{n+1}} \quad [step\ 5]$$

$$a_{n+1} \leftarrow a_n - \frac{\gamma \widehat{m}_{n+1}}{\sqrt{\hat{u}_{n+1}} + \epsilon} \quad [step\ 6]$$

The above is the iteration scheme [3.16] where, $n \in N$, $a = (W^1, g^1, W^2, g^2, \dots)$ as presented in subsection 3.2, $\big[loss\big(\varphi(x_i; a_n), y_i\big)\big]_n$ is the stochastic gradient as presented in 3.3.2, $m$ and $n$ are the first and second moment estimates respectively (from where the method takes its name), $b_1$ and $b_2 \in [0,1)$ are the exponential decay rates for the first and second moment estimate, $\widehat{m}$ and $\hat{u}$ are the bias-corrected first and second moment estimates, $b_1^n\ and\ b_2^n$ are the decay rates denoted to the power of the $n$ which is the number of the iteration the algorithm is currently on, $a > 0$ is the step size and finally $\epsilon > 0$ is a really small, steady stabilization factor which exists to prevent division with zero.

According to the original creators of Adam, as well as multiple empirical observations, the parameters of Adam typically require little to no tunning in a case-to-case basis. The parameters they recommend as default values are, $a = 0.001$, $b_1 = 0.9, b_2 = 0.999, \varepsilon = 10^{-8}$ and initialization of the estimates with $m_0 = 0$ and $u_0 = 0$. Of course, if a particular case needs some reconfiguration of the parameters described above, some trial and error will be needed to retune them to the appropriate values. In the general case though, they work fairly well [27].

Adam is very powerful optimization algorithm which outperforms its contemporaries when it comes to computational efficiency and drastically reduces training times. Of course, like other algorithms presented in this work, Adam needs to be implemented carefully and cannot be viewed a panacea against all optimization problems. Despite of its great performance, Adam performs best in really big and complex datasets and is the go-to algorithm for many applications. If the dataset though is relatively small or lacks complexity the usage of Adam, withs its multi-step iteration scheme, can actually harm the overall. Also, the type of machine learning approach implemented plays an important role in the overall performance. For example, for Neural Networks that are the focus of this work, and for a fairly big and complex datasets like the image based MNIST and CIFAR-10, the researchers behind Adams original paper found that all other factors considered steady, their method easily outperformed the best of their contemporaries as one can observe in figure 7 and figure 8 with two different architectures. The architectures referenced in figures 7 and 8, are the Multi-Layer Preceptor (MPL) and a Convolutional Neural Network (CNN) which will be explained and analyzed in the following sections of this work.

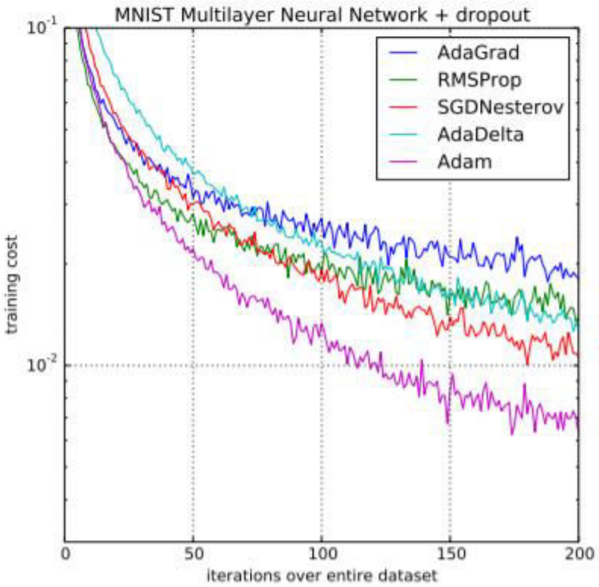


Figure 7. [27] Training cost of an MLP Neural Network on the MNIST dataset, using the Adam optimizer compared to other popular modern optimizers.

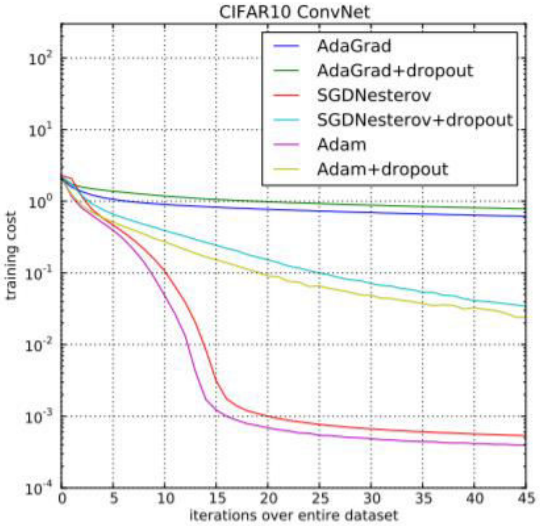


Figure 8. [27] Training cost of an CNN Neural Network on the CIFAR-10 dataset, using the Adam optimizer compared to other popular modern optimizers.

### 3.3.7 Acknowledgements of Omissions

It is important to recognize that this work neglected to mention and review a lot of fairly important optimization algorithms often used in machine learning. One especially important category of methods that this work omitted, are the Newton Method based algorithms. On these kinds of approaches, minimization of the objective function is not based on the original gradient descent method. Briefly described, one uses the second order approximation of the objective function obtained from the Taylor series of the function and, after selecting an appropriate step size, uses the gradient of the approximation to decide the direction of descent. This method can converge in fewer iterations compared to gradient descent and also guarantees a solution. Unfortunately obtaining the second order representation of the function either through numerical means (as mentioned above) or analytically could be from computationally expensive down to completely impossible. That said, there are certain cases that Newton based approaches are suitable, but in the context of this work is preferred not to formally introduce them [20]. The main reason behind this decision is that the literature review of the relevant papers about fault diagnosis of rotating machinery, revealed that gradient based approaches are more than capable to handle the problem of time-series classification and the consensus around the best optimization algorithms for fault diagnosis is heavily tilted towards gradient based methods.

It is also important to point out that the sub selection of methods presented above are by no means a thorough review of the vast field of optimization methods for machine learning problems. The goal of this section is to present the basics of the field (gradient descent, SGD, BGD), a few improvements upon this basic (GDM, AGD) and a modern approach (Adam), in order to present the ever-evolving world of machine learning optimization and how modern methods are building upon the previous work, while laying the ground for the case study and the specific methods that will be used. In order to achieve that goal, only the above methods are presented, and many other modern and more complex methods are not.

Finally, the methods presented in this section are well documented and thoroughly tested in the relevant literature. As a result of that, there are considered useful and effective so their respective convergence analysis that shows exactly why these methods work will be omitted. Such an analysis is considered beside the point of this thesis and the methods will be used as presented without additional proof of their effectiveness.

### 3.3.8 Some Observations for the Selection of an Optimization Method

Optimization algorithms are an essential part of every machine learning problem and the selection of the right one could greatly affect the efficiency, performance and success of the training procedure. However, this easier said than done since there are literary hundreds of different ones to choose from, each one accompanied by its own perks and traits. Luckily though most of the methods can be boiled down to the basics, which are the SGD and BGD methods, so one can decide which suits the problem in hand and start from there. In any case, selecting an optimization algorithm is not a trivial task and a useful tool can always be trial and error tests. Also, having extensive knowledge of the relevant literature and a fair amount of experience is essential to be able to find the methods that better work with one's case. There are no strict rules to follow when deciding optimization approaches, and chances are that more than a few can perform the task in hand, after that it is matter of each case's requirements in performance and effectiveness.

## 3.4 The Backpropagation Algorithm

As previously stated, the backpropagation algorithm is responsible for the important task of actually calculating the gradients that the optimization algorithm will use to minimize the empirical loss function. Calculating the gradient of a function might be a trivial task for a simple function with a few parameters, but even for a relatively small NN the empirical loss functions are comprised by many thousands of different parameters. As a result of that an efficient and fast algorithm needs to be introduced to perform that task. Backpropagation is by far the most used algorithm for the task, is incredible efficient on its job and will be the focus of this work on that field.

By now, a handful of different algorithms have been presented and have been layered upon each other, so it is a good time to compartmentalize the training procedure of the NN in order to simplify it and better understand it. The training of the NN can be divided into two stages, first comes the *forward pass* and the *backwards pass.* The forward pass begins the moment a training value is introduced to the first layer of the NN, then follows every transformation that is happening on this value to every other hidden layer and finishes when then value, now transformed by the effect of the layers, the activation functions and the weights/biases of each node, reaches the output layer. At this point, the backwards pass can take place. First, the backpropagation algorithm using the chain rule calculates the gradients, then these gradients are fed into the optimization algorithm that decides the direction of descent and finally the weights and biases of each node are updated, and the network is ready to perform this task again. The result of each iteration of the backwards pass, is a swift to the weights and biases of the whole NN, to a direction and magnitude that will most effectively move the objective function of the problem towards its minimum [21].

Let us review now exactly how the backpropagation algorithm handles the calculation of these huge gradients efficiently. As previously stated, backpropagation takes advantage of the chain rule the for calculation of derivatives, so defining the chain rule seems like a great place to start the analysis. Assuming $n$ functions $f_1, f_2, \dots, f_n$ differentiable, which can define a composite function $f_1 \circ (f_2 \circ \dots (f_{n-1} \circ f_n))$, then the generalized chain rule can be expressed like so.

$$\frac{df_1}{dx} = \frac{df_1}{df_2} \cdot \frac{df_2}{df_3} \cdot \dots \cdot \frac{df_n}{dx} \quad [3.17]$$

The chain rule, in practice, means that knowing the rate of change of $f_n$ relative to $x$ and every other rate of change of $\frac{df_{n-1}}{df_n}$ for every function of the convolution, then the rate of change $\frac{df_1}{dx}$ can be computed [IV]. In the context of NN this property can be extremely useful, because when the time comes during the training of a NN to calculate the gradient of the empirical loss function, which in most cases is comprised from thousands of parameters, instead of computing directly that huge gradient, one can calculate the much smaller gradients between each node starting from the back of the NN, and then multiply them to find the gradient of interest. This exactly what the backpropagation algorithm does, and it is the reason than NN are computationally manageable instead of impossible.

Since the intuitive explanation of the backpropagation algorithm has been presented, and with that understanding in mind, it can be useful to rephrase the chain rule in the context of the backpropagation algorithm. This is a good point to remind that what the actual quantity that the algorithm aims to calculate is the partial derivative of the empirical loss function in respect to the parametric vector $a$ which contains the weights and biases of the network. Mathematically, it can be represented as follows.

$$\frac{\partial L_e}{\partial a} = \sum_{l=0}^{L} [\frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial L_{e_{k,l}}}{\partial a_{n,l}} ]_l \quad [3.18]$$

Where $l$ is the index of the layer in which the algorithm currently is, $L$ is the number of all the layers. In order to calculate the right-hand term, the equation [3.19] is needed.

$$\frac{\partial L_{e_{n,l}}}{\partial a_{n,l}} = \frac{\partial z_{n,l}}{\partial a_{n,l}} \cdot \frac{\partial x_{n,l}}{\partial z_{n,l}} \cdot \frac{\partial L_{e,l}}{\partial x_{n,l}} \quad [3.19]$$

In order to calculate the result of the equation [3.19], some extra definitions need to be established. First, $z_{nl}$ is basically the right-hand term of the previously reviewed [3.3] equation.

$$z_{n,l} = \sum_{i=0}^{n} \left(a_{n,l} x_{n,l-1}\right) = \left(W_{0,l} x_{0,l-1} + g_{0,l}\right) + \left(W_{1,l} x_{1,l-1} + g_{1,l}\right) + \cdots \quad [3.20]$$

As a result of the above, it is possible to calculate all the derivates of the [3.19] equation, and the following results.

$$\frac{\partial L_e}{\partial a_{n,l}} = x_{n,l-1} \sigma'\left(z_{n,l}\right) \frac{\partial L_e}{\partial x_{n,l}} \quad [3.21]$$

Where $\sigma'$ is the derivative of the activation function. As one can observe, there is still a partial derivative on the expression [3.21], but one shall fear not because the useful tool of the chain rule, applied over the entire layer leads to the following expression.

$$\frac{\partial L_e}{\partial x_{n,l}} = \sum_{n=0}^{l-1} x_{n,l+1} \sigma'\left(z_{n,l+1}\right) \frac{\partial L_e}{\partial x_{n,l+1}} \quad [3.21]$$

Unfortunately, there is yet another derivative in [3.21] as well, but as one can see, it is for the layer $n + 1$, so one can use the expression again and again, for all the layers of the NN. The great thing though, is that the number of layers is finite and the last layer – the output layer – does not have any weights or biases and the last derivate can be calculated to be the following.

$$\frac{\partial L_e}{\partial x_{n,L}} = 2\left(x_{n,L} - y_n\right) \quad [3.22]$$

Now everything is known, and the backpropagation algorithm can finally compute the gradient and then update every weight and bias in the network. This procedure is often referred to as “learning”. On each iteration of the algorithm, for every data point of the training dataset that is, the NN is “learning” a few more things about the dataset and gets a small step towards the minimum of the objective function [IV] [28].

## 3.5 Activation functions

Activation functions, as presented in subsection 3.3, are responsible for the way each transformed sample moves from one layer of the network to the next. The selection of activation functions between the layers can have drastic impact on the training of a NN and has to be done carefully. The main reason behind their importance is that they introduce non-linearity into the network. Taking another look on the equation [3.3], it can become quite clear that without the $\sigma$ activation function this would be nothing more than a linear transformation of the receiving signals.

$$x_i^{(j)} = \left(W^j x_i^{(j-1)} + g^j\right) \quad [3.3\ without\ \sigma]$$

Practically, a NN formed from layers connected with just linear functions between them would inevitably result in a prediction function $\varphi$ that would be itself a linear function, since *the composition of linear functions is also a linear function.* That could be a problem since there is no chance that the data the NN tries to model are linear. So, in order to capture this non-linearity of the data the network is trained on, a non-linear activation function is introduced to parse the output of one layer to the next.

The selection of the exact function that will be placed in each layer is actually quite arbitrary and, like so many things in machine learning, there is no particular cookbook to follow when designing a network. Some of the most frequently used though are the following activation functions.

● The *sigmoid* function. It is one of the most used activation functions there are and maps the output of the layer in the range $(0,1)$. Sigmoid function and functions that occur from the sigmoid (e.g., the derivative of the sigmoid is also commonly used as activation function) are often used in classifiers.

$$sigmoid(x) = \frac{1}{1 + e^x} \quad [3.23]$$

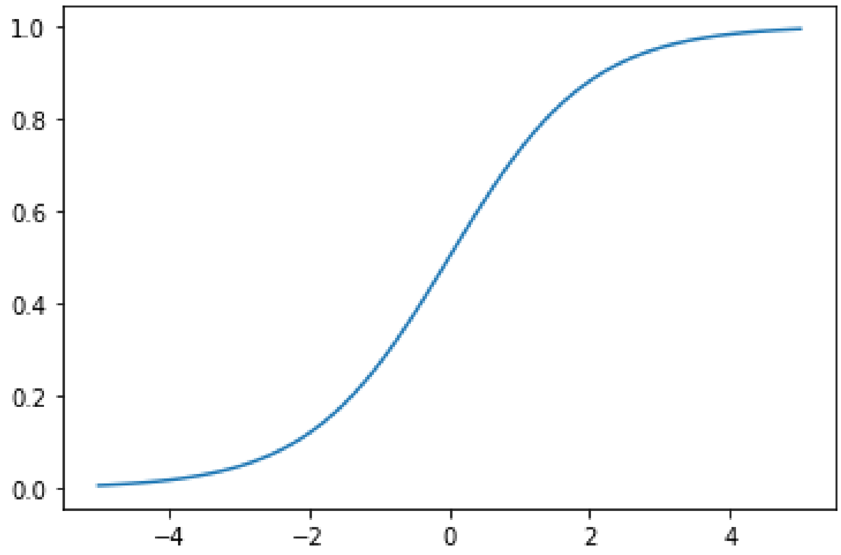


Figure 9. The sigmoid activation function.

● The *hyperbolic tangent* function. It maps the output of the layer in the range $(0,1)$, and it is most commonly used also in classifiers, but its use has been linked to the vanishing gradient problem, which can cause gradient based optimization methods to completely fail in training.

$$\tanh(\mathrm{x}) = \frac{2}{1+e^{-2x}} - 1 \quad [3.24]$$

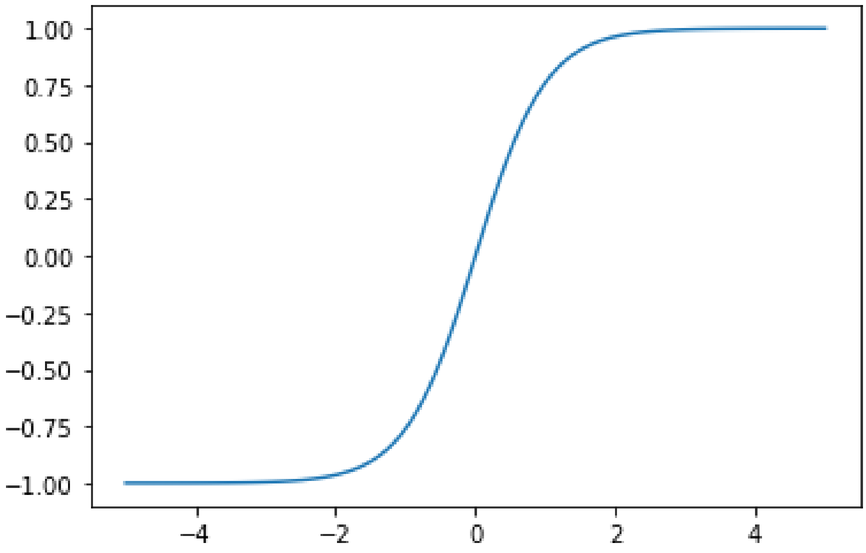


Figure 10. The tanh function.

● The *rectified linear unit function* or *ReLU* for short. This is one of the most modern and by far most used activation function. It is really computationally efficient, because if the output of the layer is smaller than zero, the neuron is not activated (i.e., just a zero is returned). At time of writing, ReLU has become pretty much the default for most cases of NN and has demonstrated itself to be really effective. ReLU is so successful that a good rule of thumb when one tries to select an activation function, is to start from ReLU and if the results are not optimal then try some other function.

$$ReLU(x) = 0, \qquad if\ x < 0$$

$$or \qquad\qquad\qquad\qquad [3.25]$$

$$ReLU(x) = x, \qquad if\ x \geq 0$$

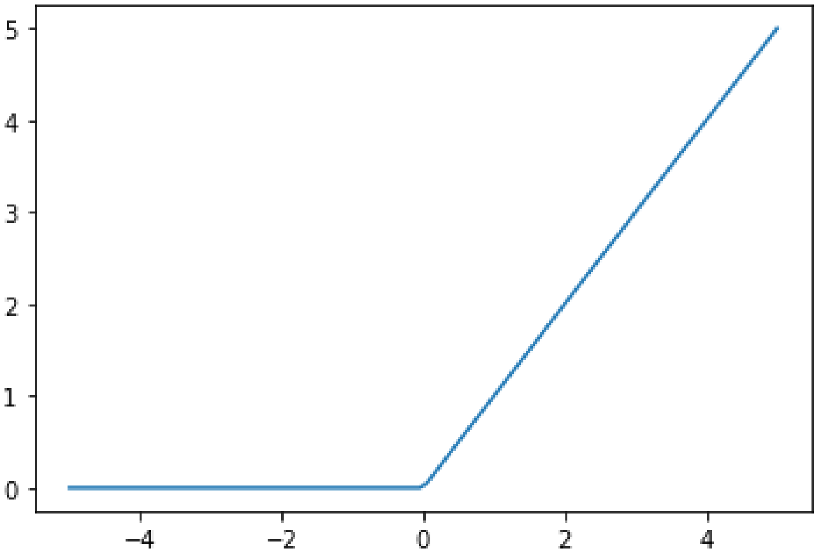


Figure 11. The ReLU function.

● The *SoftMax* function. This function is most commonly used on the last layer before the output layer for multiclass classification problems and returns the possibility of a sample to be part of each class involved in the problem.

$$SoftMax(x)_j = \frac{e^{x_j}}{\sum_{n=1}^{N} e^{x_k}} \quad [3.26]$$

Where $j = 1, \ldots, K$ the number of classes of the problem.

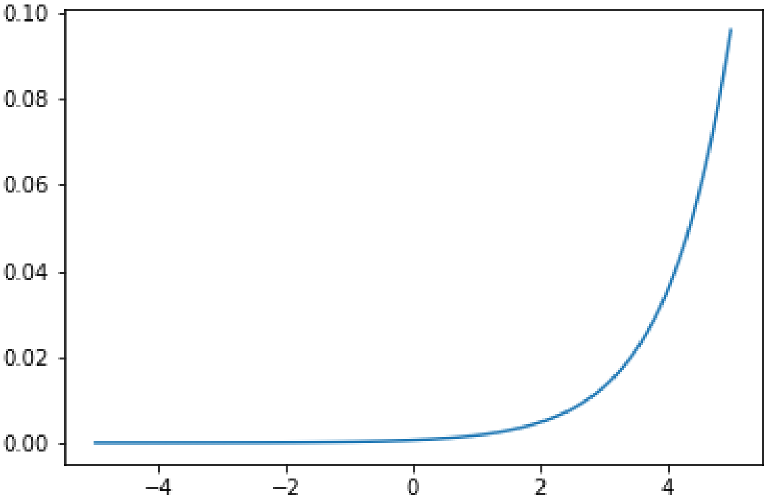


Figure 12. The SoftMax function.

## 3.6 Loss Functions

Looking back to the equation [3.6], the term $loss$ function has been used, but not yet thoroughly explained. As previously presented, $loss$ is a function which provides a continuous approximation of the cost measurements for predicting the value of $\varphi$ when actual label is $y$. To frame it in a more intuitively way, the loss function has the challenging job of trying to unsheathe the underlying effect that previous layers had on the sample and boil down to a single value which, if shrank, will indicate an improvement of the model. The described task is by no means an easy one, but extensive research on the field has presented a few different options for on to choose from when constructing a NN. Again, the selection of the right loss function has an innate arbitrariness and is based on comparative trials, as well as some experience with the field. That being said, at this point it has become pretty standard that for classification problems the best and most useful loss function is by far the cross-entropy loss.

*Cross-Entropy Loss* simply known as cross-entropy, is an approach that each predicted probability is compared to the actual output, known by the training dataset, and their comparison produces a score that penalizes the probability in accordance with its distance from the actual value [29].

To understand it, the first thing that needs a definition is entropy itself. Entropy is the measure of uncertainty of a randomly selected variable. Mathematically expressed like so.

$$H(x) = -\sum_{x} p(x)\log(x) \quad [3.27]$$

Where $H$ is the entropy, $x$ the random variable and $p$ the probability mass function of $x$.

Taking a closer look on the equation [3.27], it is revealed that the entropy is closely connected to the expectation of $x$.

$$H(x) = \mathbb{E}_{X\sim p(x)}\left[\log\left(\frac{1}{p(x)}\right)\right] \quad [3.28]$$

From [3.28] it can be concluded that the entropy of a random variable $x$ is the expected value of $\log\left(\frac{1}{p(x)}\right)$ and it can be denoted as $H(p)$.

If a distribution $q(x)$ is assumed, which describes a model's approximation of the distribution of $p(x)$, the relative entropy between $p$ and $q$ which measures their difference.

$$D(p||q) = \mathbb{E}_{X\sim p(x)}\left[\log\left(\frac{q(x)}{p(x)}\right)\right] \quad [3.29]$$

The equation that defines the cross-entropy loss distribution, denoted as $H(p, q)$ is the addition of [3.28] and [3.29].

$$H(p,q) = H(p) + D(p||q) \quad [3.30]$$

Finally, if the equation [3.30] is expanded, the final form of the cross-entropy loss distribution can be expressed by the following.

$$H(p,q) = \mathbb{E}_{X\sim p(x)}\left[\log\left(\frac{1}{q(x)}\right)\right] \quad [3.31]$$

The goal of every machine learning that employs the cross-entropy approach is to minimize this expected value described by [3.31].

## 3.7 Common Neural Network Layers and Architectures

Everything that has been presented by this point, was presented under the assumption that [3.3] is the equation that governs the transformations that take place in every layer in a NN, that assumption though is not always the case. Equation [3.3] defines a type of layer called *canonical fully connected layer* and the NN built with these kinds of layers are known as *Multilayer Perceptrons* (MLP). They are the introductory point for every analysis of how the NN work because they are simpler than different architectures, but also the scope of their success can be limited due to their simplicity. That being said, other layers and different architectures (that ones that are of interest in the context of this work at least) are based on the same analysis and use the same optimization methods, forward passes and backward passes to minimize their empirical loss function. The sole difference between the architectures that will be presented in this subsection is the transformation that happens when a sample reaches the node of the layer. It is important to note that, as simple as MLPs might be, they are still a useful tool and in certain problems can work with satisfactory results, but they certainly leave plenty of room for improvements that different layers and architectures try to address.

### 3.7.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are by a landslide the most referenced and most commonly used network architecture there is as figure 13 clearly shows.

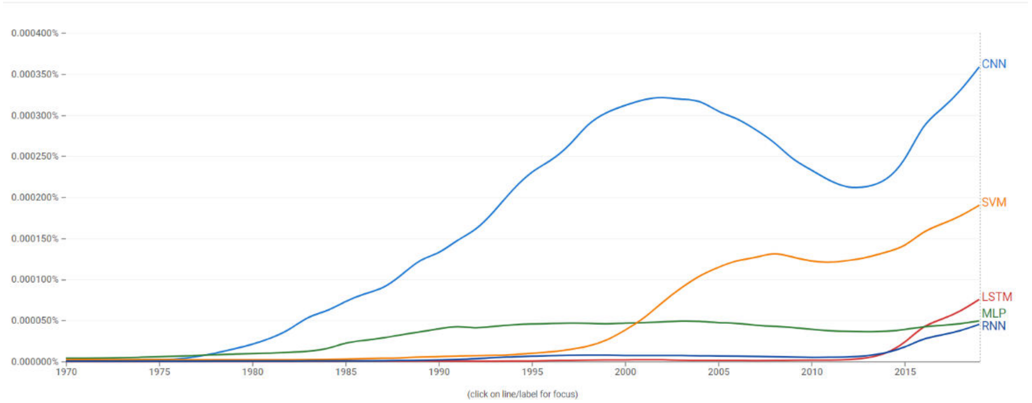


Figure 13. [VII] Comparative figure of the evolution of the number of references for CNN, SVM, LSTM, MLP and RNN in the English literature from 1970 to 2019.

CNNs were designed with the task of image classification in mind, and in that respect, they are extremely successful and are the bedrock of some of the most well-known NN on the field like AlexNet and ResNet. Their main advantage is their ability to preserve the spatial relationship between the data due to the sparse connectivity that governs the weight and biases matrix. Another biproduct of that fact is that CNNs are much more computationally cheap than their alternatives due to the small number of weights and biases that a sparse matrix need to contain. That also means that a CNN can be drastically larger in scope than any other architecture. For all those reasons CNNs are really effective in feature extraction from two-dimensional images, three-dimensional video feeds and, most importantly for this work, from one-dimensional time series data. Additionally, another important feature of CNNs is weight sharing. In CNNs, the weights are not node-specific like the ones previously reviewed on MLPs, rather they are layer-specific meaning that the whole layer shares the same weights which drastically reduces the computational burden of the model [30] [31] [32].

Typically, two kinds of layers are needed to build a CNN, the convolutional layer and the pooling layer. Usually, canonical fully connected layers are also used towards the end of the network to connect the convolutional layers with the output layer.

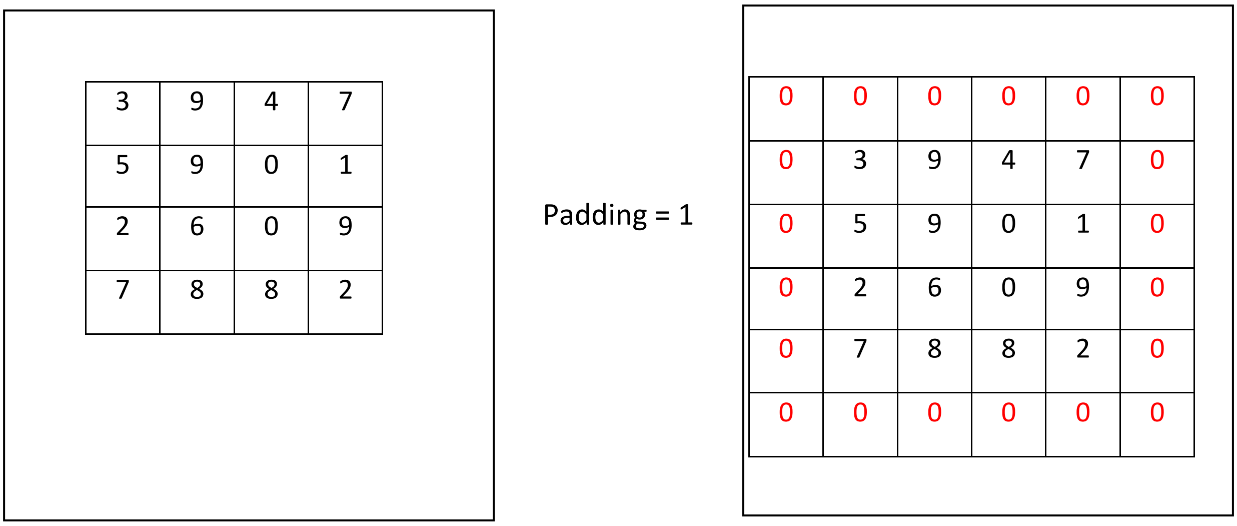
● Convolutional Layers: Arguably the most important part of the network, they are responsible with the task of performing convolutional transformations between a predetermined *filter* and the input. Filters are matrixes with predefined size, that contain the weights of the layer. Typically, the initialization filter in the beginning of the training is just an array of random numbers. In general terms, convolution is the procedure of taking the filter and performing a sliding dot operation with the entirety of a given sample signal. Mathematically this idea can be expressed like so.

$$x_k^l = \sigma(g_k^l + \sum_{i=1}^{N_{l-1}} (F_{ik}^{l-1} * x_k^{l-1})) \quad [3.32]$$

Where $k \in N$ is the number of layers in the $l \in N$ layer, $F$ is the filter containing the weights of the layer and $*$ is the convolutional operator.

The most important thing about any CNN is its filter because it is responsible what is referred to as *feature extraction*. Features are the parameters that define a data sample, and their existence or absence in the data is moving the training forward. For example, if one tries to train a NN to predict if a person will develop a disease based on their weight, height, age and BMI, those four things would be the features defining the problem. In other NN architectures, this categorization into features needs to happen manually from the NN's designer, but CNNs are able to automatically extract the features from the dataset through the use of their filter. Generally speaking, filter size heavily effects the level of detailed features extracted from the data. Larger filters tend to extract more abstract features from a dataset, while smaller ones extract more detail. For that reason, bigger filters tend to be placed earlier on the network and smaller later on. For example, if a CNN is fed an image of a person, the first layers might extract the contour of the person and the final layers the color of their eyes.

There are also two more important parameters unique in convolutional layers, padding and stride. Padding is a relic of the image recognition origins of CNNs, and its aim is to enhance the models' capabilities along the border of the image. Depending on the value of the padding, a $p$-sized frame of zeros is placed around the image, like image 8 represents. Of course, the same principle is also true for convolutional layers applied on time series data and not images, but in this context, padding is not as useful or needed and a zero-padding approach is preferred.



|  |  |  |  |
|---|---|---|---|
| 3 | 9 | 4 | 7 |
| 5 | 9 | 0 | 1 |
| 2 | 6 | 0 | 9 |
| 7 | 8 | 8 | 2 |

Padding = 1

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 3 | 9 | 4 | 7 | 0 |
| 0 | 5 | 9 | 0 | 1 | 0 |
| 0 | 2 | 6 | 0 | 9 | 0 |
| 0 | 7 | 8 | 8 | 2 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |

Image 8. Example of $padding = 1$ for a random two-dimensional matrix.

Stride is the step that the sliding filter is making on the data sample. Larger strides have the effect of lowering the dimension of the layers output which can lighten the computational error, but in expense to networks effectiveness since bigger steps means less information to extract features from. Tunning this parameter usually means trial and error testing. An example can be seen in image 9.

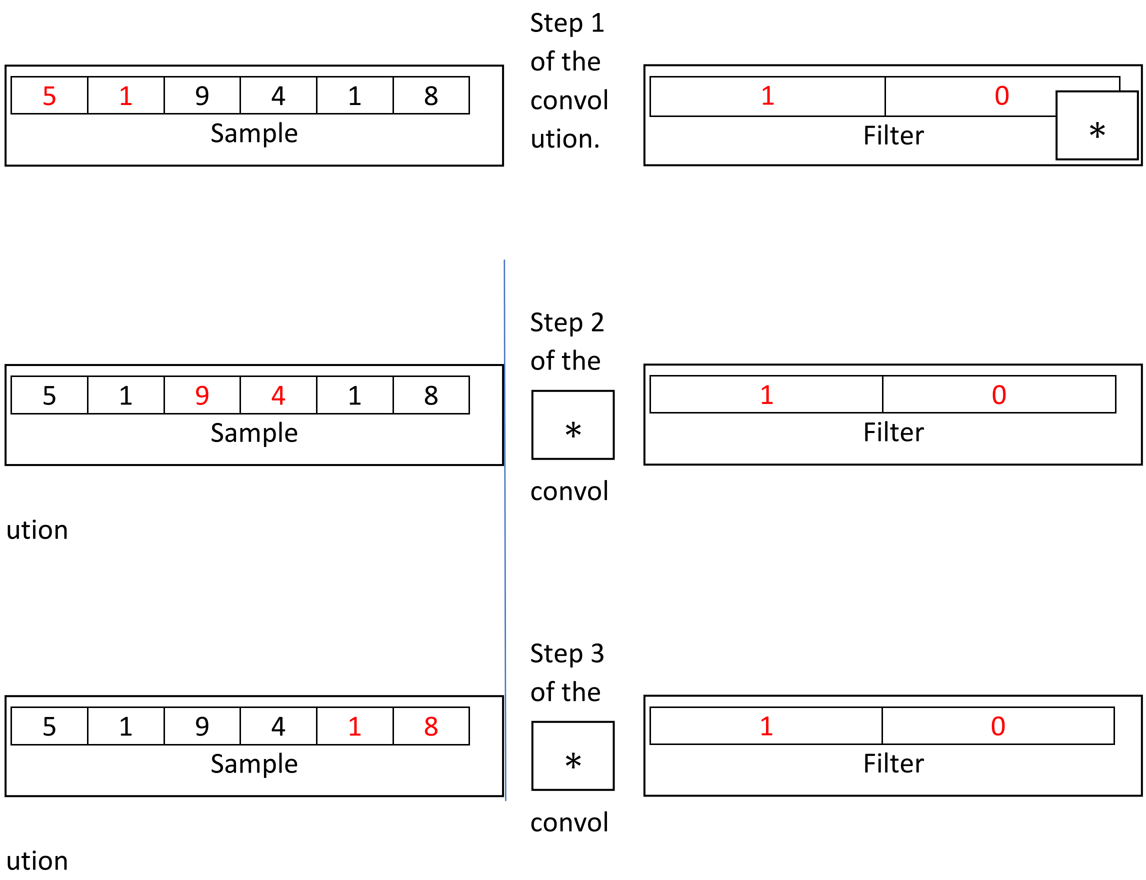


Image 9. Example of $stride = 2$ for random matrixes. The red numbers represents the part of the sample that convolutes with the filter in each step.

The dimension $d'$ of the output of every layer can be calculated from the following formula.

$$d' = \frac{d - f + p}{s} + 1 \quad [3.33]$$

Where $d$ is the dimension of the input, $f$ is the size of the filter, $p$ is the padding and $s$ is the stride.

● Pooling layers: They are placed after a convolutional layer and their purpose is to subsample the output of the convolutional layer. This accomplishes two things, first it scales down the transformed samples, thus further adding to the computational efficiency of CNNs and secondly due to the way the subsampling is performed, it preserves only the most pronounced features of the sample. Pooling is performed by specifying the dimension of the pooling region on the sample and a stride parameter, in similar fashion to the convolution. There are different types of pooling layers, like max pooling, min pooling, average polling, etc. The most commonly used though is arguably max pooling, which looks at the specified region and only transports its biggest value to the next layer.

Pooling layers do not need an activation function, since they do not perform a linear transformation to the sample, they just cherry-pick the most important values of the sample.

The problem with pooling arises when the information cut from proceeding through the network is more that it should, thus bringing down the overall performance of the network. This occurs when the pooling region is greater than it should.

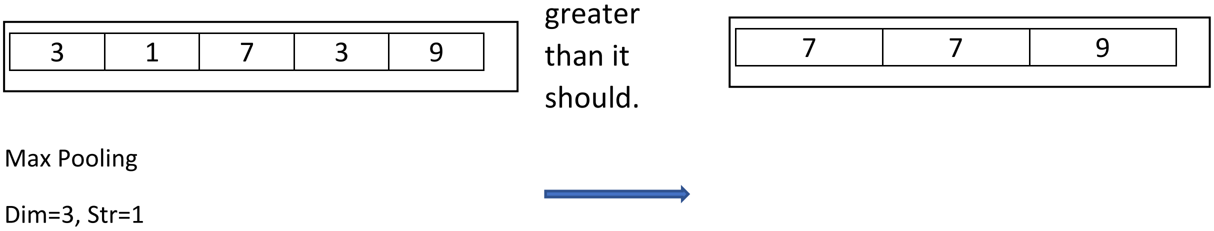


Image 10. Example of Max Pooling with a pooling filter of dimension=3 and a stride=1.

The dimension $d'$ of the output of every layer can be calculated from the following formula.

$$d' = \frac{d-f}{s} \quad [3.34]$$

Where $d$ is the dimension of the input, $f$ is the size of the pooling filter and $s$ is the stride.

### 3.7.2 Multilayer Perceptrons

Also frequently referred to simple as Artificial Neural Networks (ANNs), MLPs are one the fundamental NN structures and they utilize the fully connected layers that were presented in subsection 3.2. They are much simpler than CNNs and they utilize just one type of layer, the fully connected. One of their defining characteristics is that feature selection is not done by the network itself, like a CNN does, and the user needs to manually define the features of the dataset for the MLP to use. This is also its biggest drawback, since its fast and easy implementation make it an appealing option for classification problems [21].

## 3.8 Other Supervised Machine Learning Algorithms: SVMs

Support Vector Machines (SVM) are another supervised machine learning algorithm distinctly different from NN, but still following the same general principles of training on dataset and optimizing an objective function in order to perform predictions for unencountered data. SVM's exist for a while, in many different form factors, improvements and differentiations, but in the context of this work the focus will be on multi-dimensional SVM's because of their proved efficiency and effectiveness on binary classification problems.

Intuitively explained, the main goal of an SVM is to calculate a hyperplane which optimally separates the classes on which a dataset is divided to. Hyperplanes are subspaces of the defining space of the problem, whose dimension is equal to the defining space minus one. So, for a two-dimensional classification problem for example, the hyperplane is a one-dimensional line (or curve depending on the kind of SVM). This line separates the space into two subspaces which, hopefully, they will be occupied by the datapoints of each class, respectively. Two more hyperplanes are being defined by the points closest to the first line on each side of the space, those new hyperplanes are the *support vectors* which give the algorithm its name. They define the margins of the SVM, and they help the SVM to find the actual optimal hyperplane to separate the two classes out of many possibilities [20]. The example of a two-dimensional SVM can be seen in image 11.

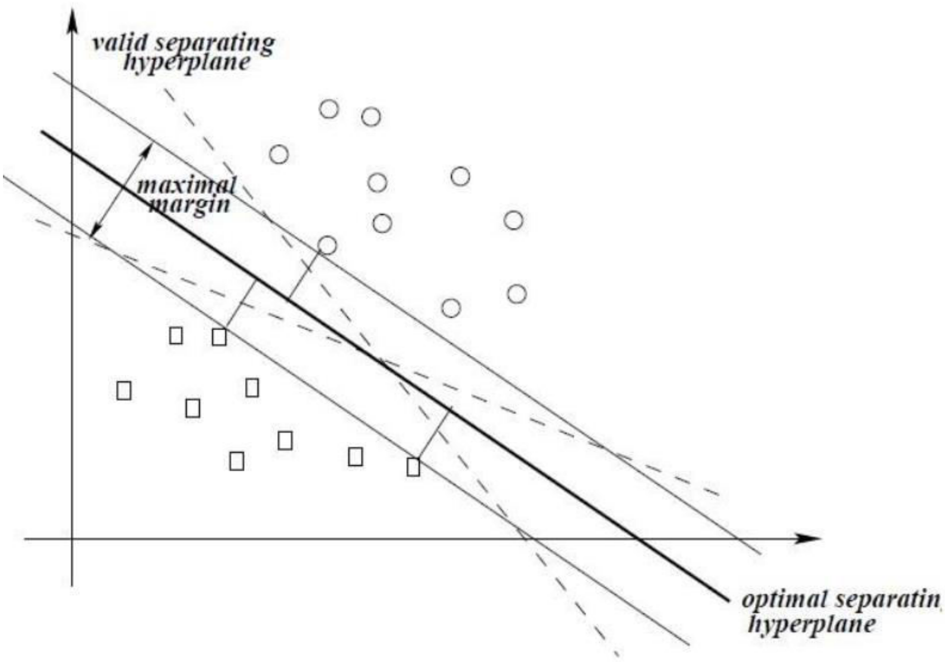


Image 11. [20] Example of two-dimensional linear SVM.

The training of an SVM adheres to the same principles as a NN, an objective function needs to be optimized and loss functions are employed to compute the error during training. Unlike NN, there is no need for different layers, and SVM's are dependent just on one transformation on the input data, thus eliminating the need for multiple layers and by extend the backpropagation algorithm.

The main boundary of the SVM can be formally expressed like so.

$$w \cdot x + b = 0 \quad [3.35]$$

Where $w$ is the vector defining the boundary, $x$ is the input vector and $b$ is a scalar threshold. Following the same logic, the upper and lower margins defined by the support vectors can be expressed accordingly.

$$w \cdot x + b = 1 \quad [3.36]$$

For the upper margin.

$$w \cdot x + b = -1 \quad [3.37]$$

For the lower margin. As a result of the right hand side parts of equations [3.35]-[3.37], the prediction function of a two-dimensional linear SVM can be formally expressed by the following expression.

$$\varphi(x) = sing\big((w \cdot x) + b\big) \quad [3.38]$$

Where $\varphi$ is the prediction function, $w$ is the vector that optimally defines the boundary-hyperplane between the classes, $x$ is the input data, $b$ is a scalar threshold and finally $sign$ is the function defined below.

$$sing(x) = \begin{cases} -1 & if\ x < 0 \\ 0 & if\ x = 0 \\ 1 & if\ x > 0 \end{cases} \quad [3.39]$$

A fair question though is how the optimal $w$ vector is calculated. For that, some constrained quadratic optimization should be employed, and the solution of the following problem is the optimal $w$.

$$minimize\{\, \tau(w) = \frac{1}{2}\|w\|^2 \,\}$$

$$[3.40]$$

$$subject\ to: \ y_i\big((w \cdot x_i) + b\big) \geq 1, \ \ i = 1,2, \dots, n$$

Where $y$ are the labels of the dataset and $n$ is the total number of training sets.

The solution obtained if the problem [3.40] gets resolved, should look like the following statement.

$$w = \sum v_i \cdot s_i \quad [3.41]$$

Where $s$ are the support vectors obtained by the training, and $v$ are parameters that act like weights and determine which input vectors are actually the support vectors, so they are defined in $v \in [0, \infty]$. As a result, the equation [3.38] can be restated like so.

$$\varphi(x) = sing\left(\sum_{i=1}^{n} v_i (x \cdot x_i) + b\right) \quad [3.42]$$

Unfortunately, the capabilities of a linear approach are quite limited, since there are many classification problems that their dataset is comprised of data points of higher dimension vectors. The generalization of the problem to a non-linear one though, is not such a challenging task since the backbone of the method remains the same. One popular way to perform that transformation is to select a kernel function which can transform a non-linear separable input space to a linear one.

$$K(x,y) = t(x) \cdot t(y) \quad [3.43]$$

Where $K$ is the kernel function, while $t$ is a function that transforms its input from a N-dimensional space to a Q-dimensional one $\mathcal{R}^n \rightarrow \mathcal{R}^Q$. So, the equation [3.42] can be now written like so.

$$\varphi(x) = sing\left(\sum_{i=1}^{n} v_i K(x \cdot y) + b\right) \quad [3.42]$$

There are a few options when selecting a kernel function, with the most common being polynomials, the sigmoid function or radial basis function [9] [33] [34].

## 3.9 Chapter Conclusions

As it has been clearly established, designing and using a machine learning apparatus is not a trivial task. A lot of things are left up to comparative trials to find the optimal decision, from optimization methods to activation functions. This is probably the most significant challenge facing the development of robust machine learning detection systems, but it is not an insurmountable obstacle. The most valuable tools when trying to take these decisions are a thorough knowledge of the relevant scientific literature and the trial-and-error tests.

In this chapter an elementary but thorough review of some of the most important machine learning algorithms has been done. The focus of this work was on classification since this is the most relevant to the context of the thesis problem that machine learning methods are able to tackle. The different approaches that has been presented – MLPs, CNNs and SVMs – are just a sub selection of many different methods, but they form the basis of the most used classification for intelligent fault diagnosis of rotating mechanical machinery one can find in the relevant literature. It should be pointed out that several things regarding NN and SVMs has been omitted. This choice was made to keep the discussion on this work within the bounds of fault diagnosis and to present only what is relevant to that context. This work is by no means a thorough review of the vast field of machine learning and its goal is to utilize some of the tools find there to achieve an approach to the much more focused problem of rotating machine fault diagnosis.

That being said, some very useful techniques has been presented by this point and the ground has been properly laid to test the approaches presented in a case study, compare them to each other and to traditional methods of fault detection and finally evaluate their performance.

# Chapter 4. Case Study

## 4.1 Introduction

By this point a few different approaches in the context of vibrational time series classification have been presented in the previous chapters. Everything presented before shall be put to test in this chapter in a case study on rolling bearing fault detection. An implementation of the traditional methodology, one MLP architecture, two different CNN architectures and an SVM will be tasked with the classification problem of diagnosing whether a rolling bearing is healthy, and if not to recognize their respective location specific defect. To do that, the publicly available rolling bearings defects dataset provided by the CWRU, the python programming language, MATLAB's programming environment and the Keras machine learning library will be employed.

## 4.2 Presenting the Problem of Rolling Bearings Fault Diagnosis

Rolling bearings are an extremely important piece of any mechanical system with moving or rotating parts. Their main role in a system is to reduce the friction between two parts of a system and allow their combined movement to take place. Damaged or broken rolling bearings can cause destructive failures for the systems they are part of, since the high friction can cause overheating problems, or even complete shutdown of the system due to inability of movement. Indicative of the importance of such components is that failed bearings are to blame for about 70% of the gearbox failures in mechanical systems [12]. As a result of the above, accurately and timely detecting bearings that are defected in order to replace them before they cause failure of the system is crucial. That way plant floors and research labs can maintain their equipment more efficiently which can save time, money and protect the bodily integrity of any personnel that manages the monitored system.

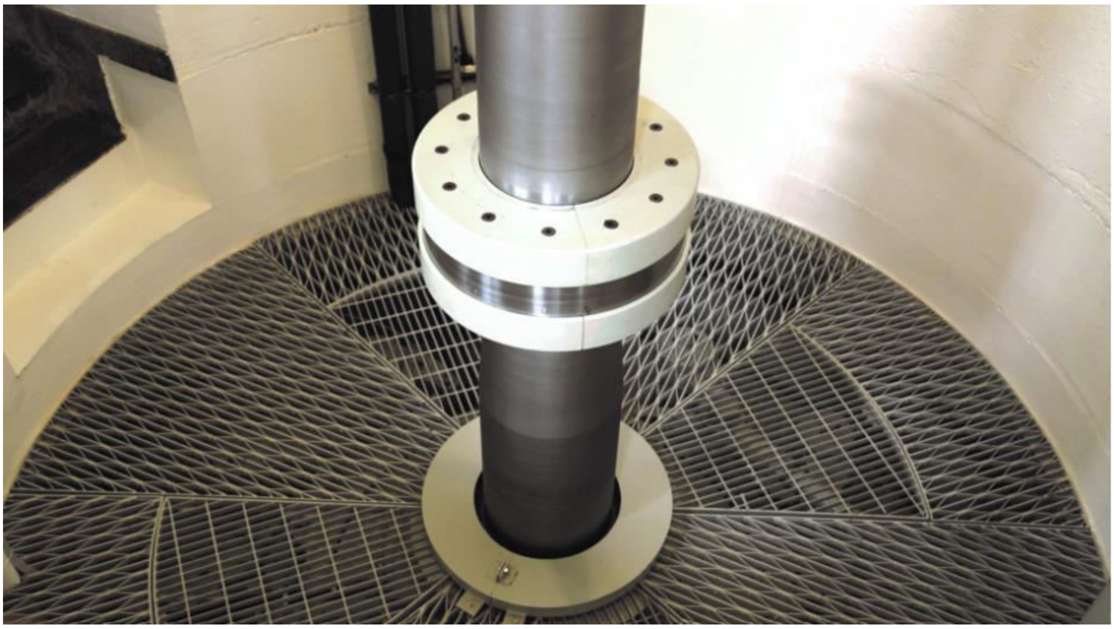


Image 12. [C] Rolling bearing in a hydro powered generator's shaft.

As it has already been presented in the subsection 2.3, when a local fault exists in a rotating mechanical component – in this case a rolling bearing – a unique vibrational signature is produced because of its rotation. These vibrational signatures resonate in specific frequencies that can be calculated from case specific empirical relations. In particular for the case of rolling bearings, three main fault frequencies can be defined, depending on the location of the fault on the bearing. These frequencies are depended on the geometrical characteristics of the bearing as well as the $f_{rm}$ rotational speed (in Hz) of the component they are attached to. These characteristics are the pitch diameter represented as $PD$, the ball diameter represented as $BD$ and the number of balls that exists in the bearing represented by $n$ [35].

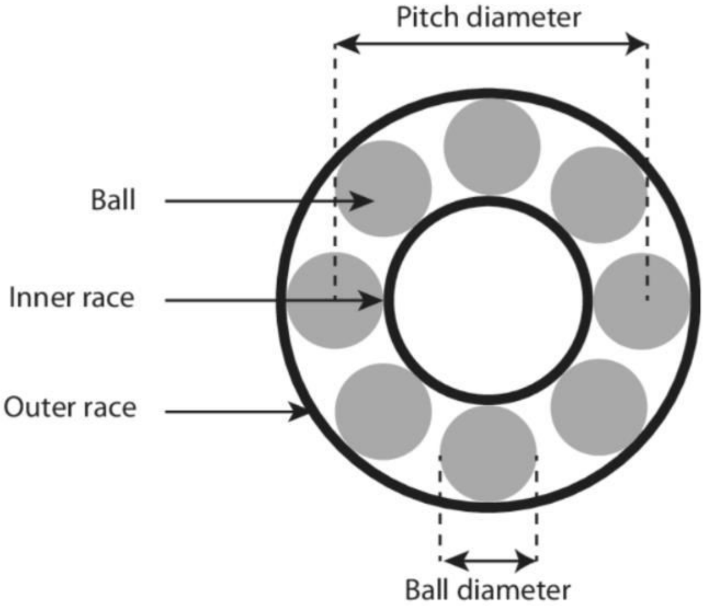


Image 12. Schematic representation of a rolling bearing and its characteristics.

- Outer race fault. It is a fault that effects the outer part of the rolling bearing, where the upper part of the balls are in touch with while they roll. Its fault specific frequency can be calculated by the following relation.

$$f_{OF} = \frac{n}{2} f_{rm} \left(1 - \frac{BD}{PD}\right) \quad [4.1]$$

- Inner race fault. It is a fault that effects the inner part of the rolling bearing, where the bottom part of the balls are in touch with while they roll. Its fault specific frequency can be calculated by the following relation.

$$f_{IF} = \frac{n}{2} f_{rm} \left(1 + \frac{BD}{PD}\right) \quad [4.2]$$

- Ball fault. It is a fault that effects one or more of the balls inside the bearing's races. Its fault specific frequency can be calculated by the following relation.

$$f_{BF} = \frac{PD}{2BD} f_{rm} \left(1 - \left(\frac{BD}{PD}\right)^2\right) \quad [4.3]$$

As a result of the above, any rolling bearing can exist in one the following states: healthy state, with inner race fault, with outer race fault or with ball fault. So, the problem of detecting faulty rolling bearing and localize their defect can be recontextualized as a classification problem. Depending on the existence of particular fault frequencies in the vibrational signals sampled from operating bearings, one can determine in which of the predefined state-classes the bearing belongs to.

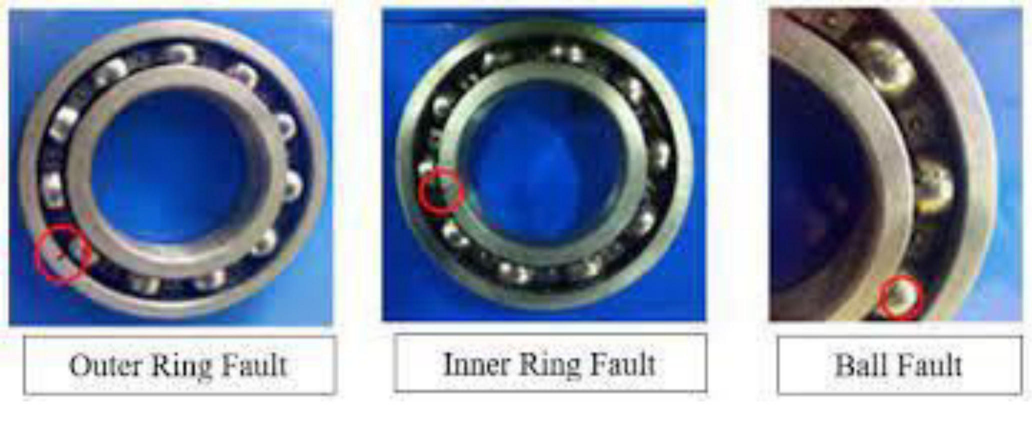


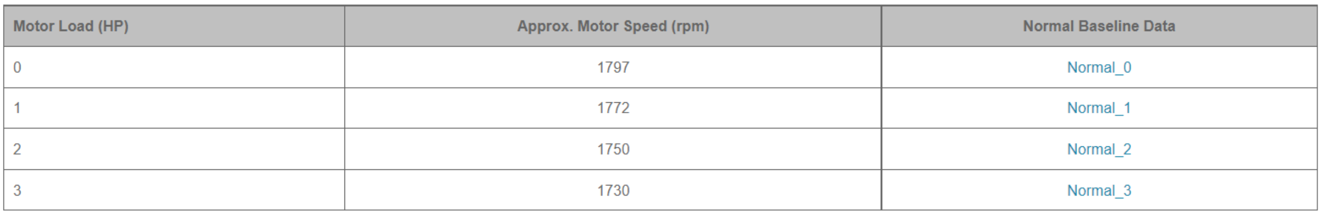
Image 13. [26] Bearings with local the three local faults described in this work.

In the context of this work, five different approaches will be used to tackle this classification problem. First, a version of the traditional envelope spectrum analysis will be reviewed, followed by a MLP approach, two different CNN architectures and finally an SVM approach.

## 4.3 Presenting the CWRU Rolling Bearing Dataset

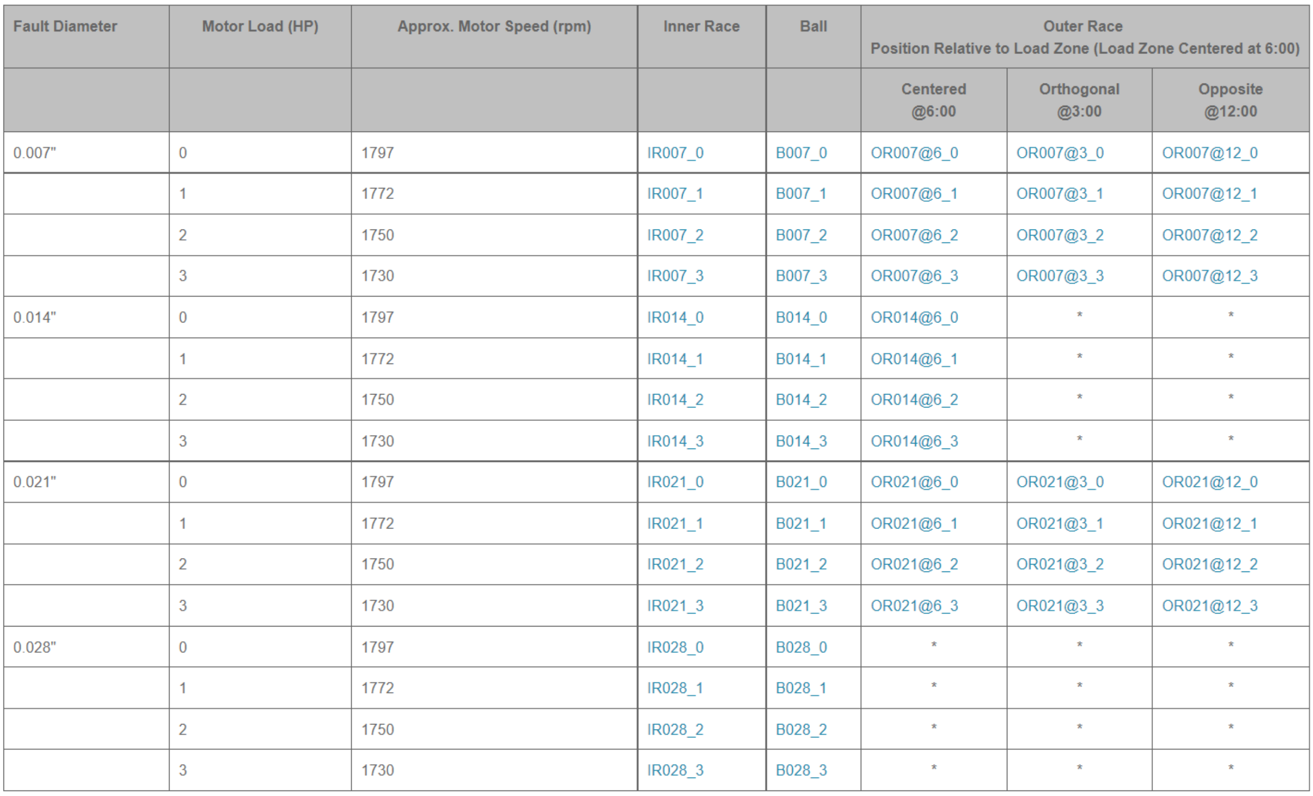
One rather important part of any attempt to review a heavily data driven problem like the one currently reviewed in this work, is the actual data that will be used to test out these different approaches. The data that will be used in this work are kindly provided by the Case Western Reserve University as a publicly available dataset of real vibrational measurements, sampled from a variety of rolling bearings, in different rotational speeds, different motor loads and with different sampling rates. The excellent and multivariant work that CWRU has done and made public for anyone to use, has made this particular dataset the golden standard and the starting point for any research done on the field.

The test rig used by CWRU to obtain the signals consists of a 2 hp Reliance Electric Motor, a torque transcoder to accurately measure the operating rotational speed, a dynamometer to accurately measure the load from the motor to the bearing, SFK rolling bearings used to support the motor's shaft, accelerometers attached to the housing of the bearings using magnetic bases and a 16-channel data logger connected to a computer to collect the obtained data like the subsections 2.1 - 2.3 demonstrated. Several different tests were performed, under different conditions, which led to the following tables with data under different combinations of motor load, fault diameter and rotational speed of the shaft. Table 2 shows the benchmark data, obtained by healthy bearings under different conditions, Table 3 was obtained under a sampling rate of 12kHz, while Table 4 under a sampling rate of 48kHz.



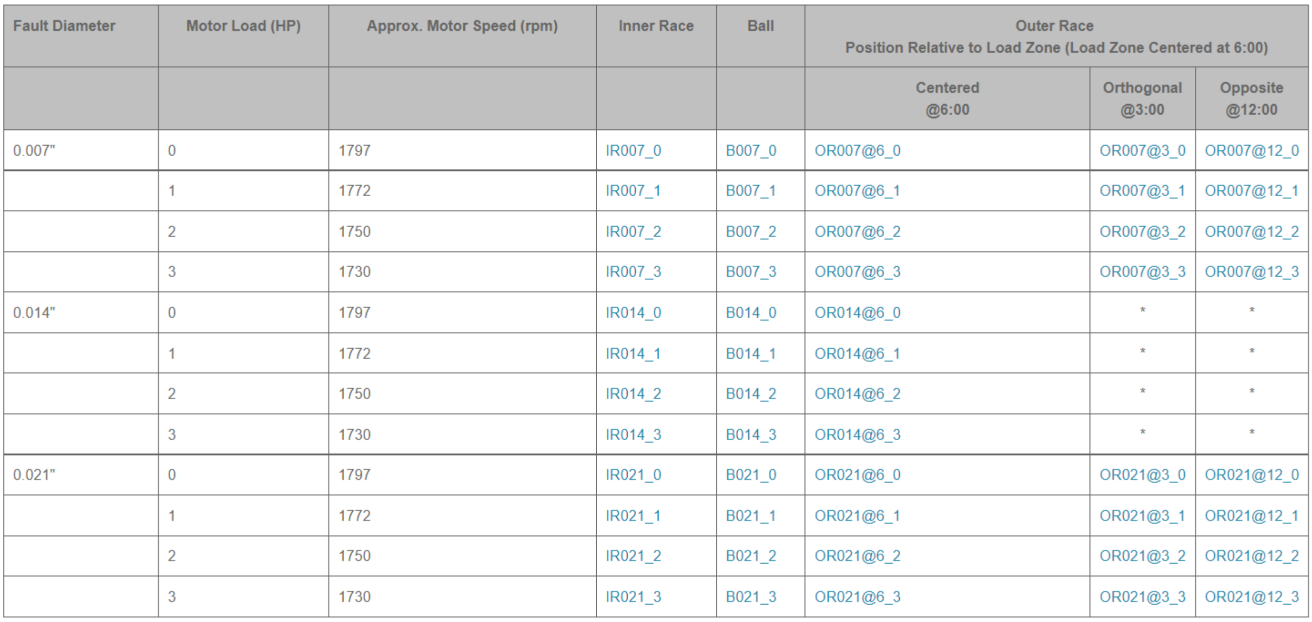
| Motor Load (HP) | Approx. Motor Speed (rpm) | Normal Baseline Data |
|---|---|---|
| 0 | 1797 | Normal_0 |
| 1 | 1772 | Normal_1 |
| 2 | 1750 | Normal_2 |
| 3 | 1730 | Normal_3 |

Table 2. [III] Different condition specific vibrational measurements for healthy bearings (CWRU).



| Fault Diameter | Motor Load (HP) | Approx. Motor Speed (rpm) | Inner Race | Ball | Outer Race<br>Position Relative to Load Zone (Load Zone Centered at 6:00) | | |
|---|---|---|---|---|---|---|---|
| | | | | | Centered<br>@6:00 | Orthogonal<br>@3:00 | Opposite<br>@12:00 |
| 0.007" | 0 | 1797 | IR007_0 | B007_0 | OR007@6_0 | OR007@3_0 | OR007@12_0 |
| | 1 | 1772 | IR007_1 | B007_1 | OR007@6_1 | OR007@3_1 | OR007@12_1 |
| | 2 | 1750 | IR007_2 | B007_2 | OR007@6_2 | OR007@3_2 | OR007@12_2 |
| | 3 | 1730 | IR007_3 | B007_3 | OR007@6_3 | OR007@3_3 | OR007@12_3 |
| 0.014" | 0 | 1797 | IR014_0 | B014_0 | OR014@6_0 | * | * |
| | 1 | 1772 | IR014_1 | B014_1 | OR014@6_1 | * | * |
| | 2 | 1750 | IR014_2 | B014_2 | OR014@6_2 | * | * |
| | 3 | 1730 | IR014_3 | B014_3 | OR014@6_3 | * | * |
| 0.021" | 0 | 1797 | IR021_0 | B021_0 | OR021@6_0 | OR021@3_0 | OR021@12_0 |
| | 1 | 1772 | IR021_1 | B021_1 | OR021@6_1 | OR021@3_1 | OR021@12_1 |
| | 2 | 1750 | IR021_2 | B021_2 | OR021@6_2 | OR021@3_2 | OR021@12_2 |
| | 3 | 1730 | IR021_3 | B021_3 | OR021@6_3 | OR021@3_3 | OR021@12_3 |
| 0.028" | 0 | 1797 | IR028_0 | B028_0 | * | * | * |
| | 1 | 1772 | IR028_1 | B028_1 | * | * | * |
| | 2 | 1750 | IR028_2 | B028_2 | * | * | * |
| | 3 | 1730 | IR028_3 | B028_3 | * | * | * |

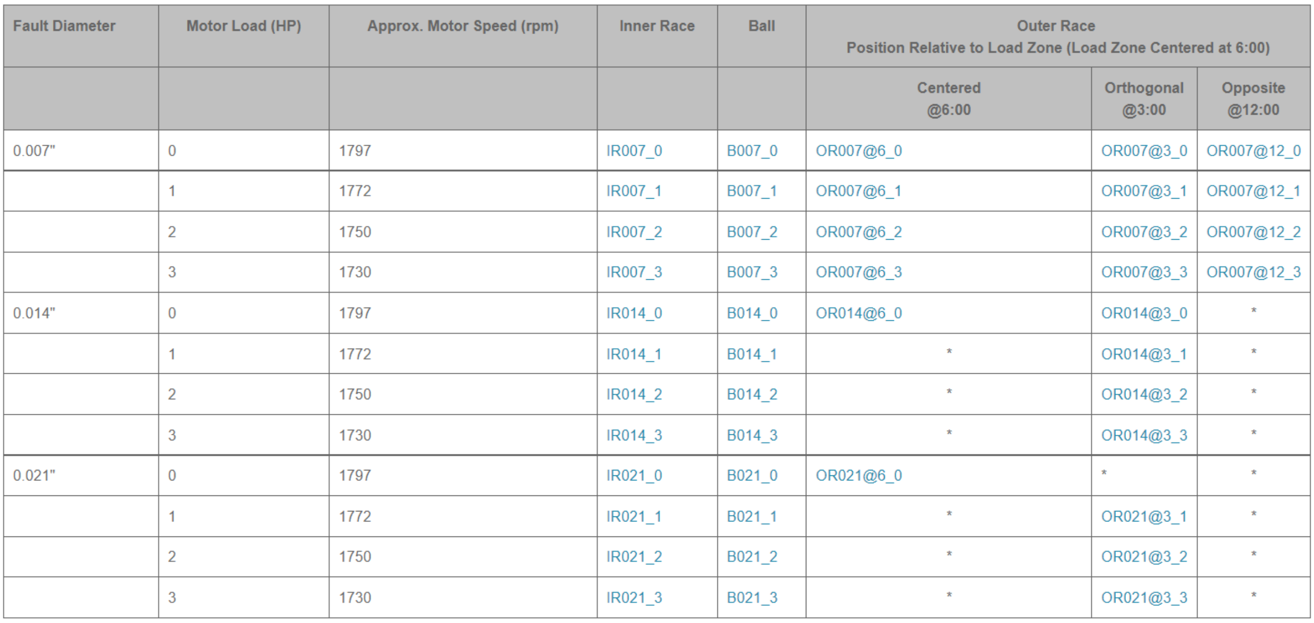
Table 3. [III] Different condition specific vibrational measurements for faulty bearings placed in the drive end of the motor under a sampling rate of 12kHz (CWRU).



| Fault Diameter | Motor Load (HP) | Approx. Motor Speed (rpm) | Inner Race | Ball | Outer Race<br>Position Relative to Load Zone (Load Zone Centered at 6:00) | | |
|---|---|---|---|---|---|---|---|
| | | | | | Centered<br>@6:00 | Orthogonal<br>@3:00 | Opposite<br>@12:00 |
| 0.007" | 0 | 1797 | IR007_0 | B007_0 | OR007@6_0 | OR007@3_0 | OR007@12_0 |
| | 1 | 1772 | IR007_1 | B007_1 | OR007@6_1 | OR007@3_1 | OR007@12_1 |
| | 2 | 1750 | IR007_2 | B007_2 | OR007@6_2 | OR007@3_2 | OR007@12_2 |
| | 3 | 1730 | IR007_3 | B007_3 | OR007@6_3 | OR007@3_3 | OR007@12_3 |
| 0.014" | 0 | 1797 | IR014_0 | B014_0 | OR014@6_0 | * | * |
| | 1 | 1772 | IR014_1 | B014_1 | OR014@6_1 | * | * |
| | 2 | 1750 | IR014_2 | B014_2 | OR014@6_2 | * | * |
| | 3 | 1730 | IR014_3 | B014_3 | OR014@6_3 | * | * |
| 0.021" | 0 | 1797 | IR021_0 | B021_0 | OR021@6_0 | OR021@3_0 | OR021@12_0 |
| | 1 | 1772 | IR021_1 | B021_1 | OR021@6_1 | OR021@3_1 | OR021@12_1 |
| | 2 | 1750 | IR021_2 | B021_2 | OR021@6_2 | OR021@3_2 | OR021@12_2 |
| | 3 | 1730 | IR021_3 | B021_3 | OR021@6_3 | OR021@3_3 | OR021@12_3 |

Table 4. [III] Different condition specific vibrational measurements for faulty bearings placed in the drive end of the motor under a sampling rate of 48kHz (CWRU).

The tables 3 and 4 were obtained for bearings placed in the drive end of the motor, but the CWRU also made experiments for bearings placed in the fan end of the motor with a sampling rate of 12kHz. These data, according to their specific conditions can be found in table 5.

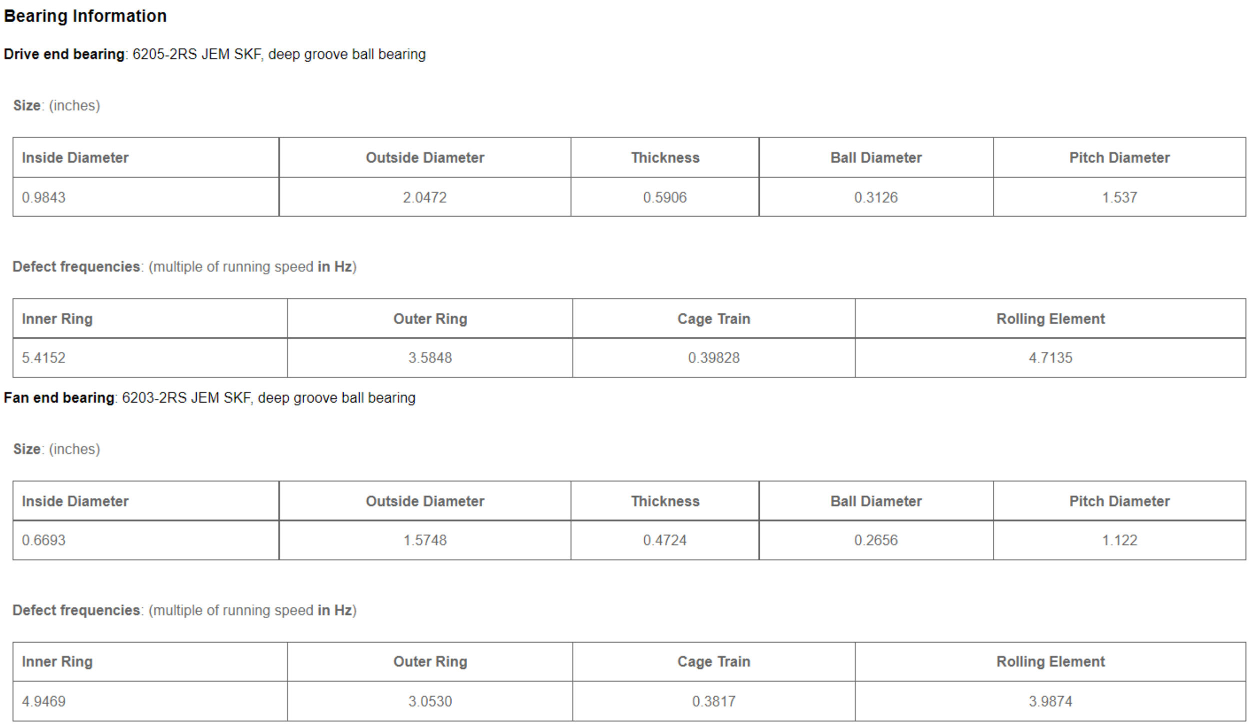


| Fault Diameter | Motor Load (HP) | Approx. Motor Speed (rpm) | Inner Race | Ball | Outer Race<br>Position Relative to Load Zone (Load Zone Centered at 6:00) | | |
|---|---|---|---|---|---|---|---|
| | | | | | Centered<br>@6:00 | Orthogonal<br>@3:00 | Opposite<br>@12:00 |
| 0.007" | 0 | 1797 | IR007_0 | B007_0 | OR007@6_0 | OR007@3_0 | OR007@12_0 |
| | 1 | 1772 | IR007_1 | B007_1 | OR007@6_1 | OR007@3_1 | OR007@12_1 |
| | 2 | 1750 | IR007_2 | B007_2 | OR007@6_2 | OR007@3_2 | OR007@12_2 |
| | 3 | 1730 | IR007_3 | B007_3 | OR007@6_3 | OR007@3_3 | OR007@12_3 |
| 0.014" | 0 | 1797 | IR014_0 | B014_0 | OR014@6_0 | OR014@3_0 | * |
| | 1 | 1772 | IR014_1 | B014_1 | * | OR014@3_1 | * |
| | 2 | 1750 | IR014_2 | B014_2 | * | OR014@3_2 | * |
| | 3 | 1730 | IR014_3 | B014_3 | * | OR014@3_3 | * |
| 0.021" | 0 | 1797 | IR021_0 | B021_0 | OR021@6_0 | * | * |
| | 1 | 1772 | IR021_1 | B021_1 | * | OR021@3_1 | * |
| | 2 | 1750 | IR021_2 | B021_2 | * | OR021@3_2 | * |
| | 3 | 1730 | IR021_3 | B021_3 | * | OR021@3_3 | * |

Table 5. [III] Different condition specific vibrational measurements for faulty bearings placed in the fan end of the motor under a sampling rate of 12kHz (CWRU).

Obviously to reliable test and compare the methods selected in this work, the data used should be sampled under the same conditions. So, in the context of this work, will be used data obtained from bearings placed to the drive end of the motor, with a fault of 0.007 inches in diameter, under a motor load of 0hp, a rotational speed of the shaft at 1797rpm and a sampling rate of 12kHz will be used, with a benchmark of normal bearings sampled at 0hp motor load, a rotational speed of the shaft at 1797rpm and a sampling rate of 12kHz. All the data are downloadable MATLAB files.

The CWRU also provides detailed information for the exact type of bearings used in the tests and their respective fault specific frequencies, seen in table 6 as well as information for the faults of the bearings, seen in table 7.



**Bearing Information**

**Drive end bearing**: 6205-2RS JEM SKF, deep groove ball bearing

**Size**: (inches)

| Inside Diameter | Outside Diameter | Thickness | Ball Diameter | Pitch Diameter |
|---|---|---|---|---|
| 0.9843 | 2.0472 | 0.5906 | 0.3126 | 1.537 |

**Defect frequencies**: (multiple of running speed **in Hz**)

| Inner Ring | Outer Ring | Cage Train | Rolling Element |
|---|---|---|---|
| 5.4152 | 3.5848 | 0.39828 | 4.7135 |

**Fan end bearing**: 6203-2RS JEM SKF, deep groove ball bearing

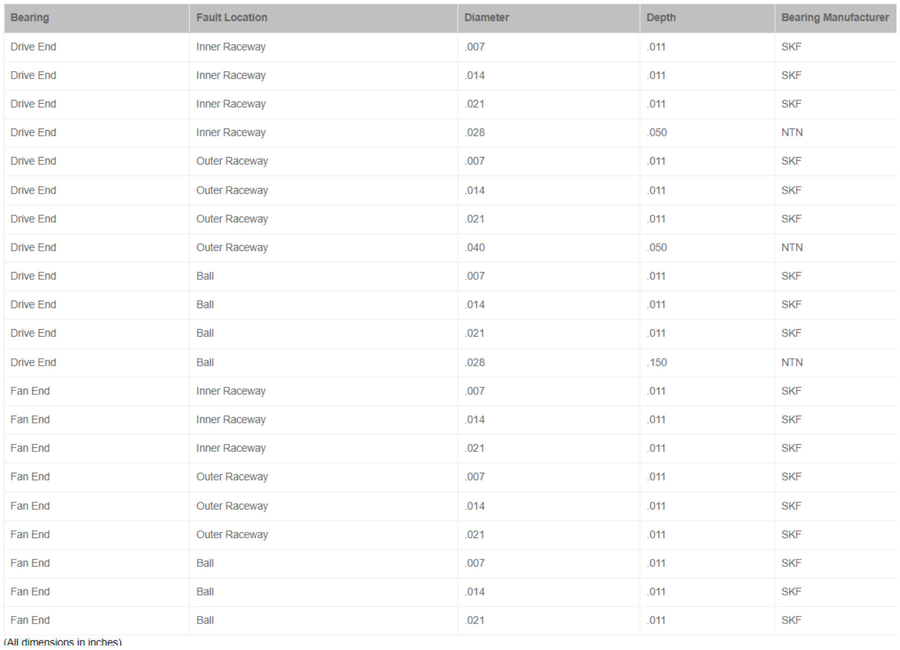
**Size**: (inches)

| Inside Diameter | Outside Diameter | Thickness | Ball Diameter | Pitch Diameter |
|---|---|---|---|---|
| 0.6693 | 1.5748 | 0.4724 | 0.2656 | 1.122 |

**Defect frequencies**: (multiple of running speed **in Hz**)

| Inner Ring | Outer Ring | Cage Train | Rolling Element |
|---|---|---|---|
| 4.9469 | 3.0530 | 0.3817 | 3.9874 |

Table 6. [III] Rolling bearings used in CWRU’s test and their geometric characteristics.



| Bearing | Fault Location | Diameter | Depth | Bearing Manufacturer |
|---|---|---|---|---|
| Drive End | Inner Raceway | .007 | .011 | SKF |
| Drive End | Inner Raceway | .014 | .011 | SKF |
| Drive End | Inner Raceway | .021 | .011 | SKF |
| Drive End | Inner Raceway | .028 | .050 | NTN |
| Drive End | Outer Raceway | .007 | .011 | SKF |
| Drive End | Outer Raceway | .014 | .011 | SKF |
| Drive End | Outer Raceway | .021 | .011 | SKF |
| Drive End | Outer Raceway | .040 | .050 | NTN |
| Drive End | Ball | .007 | .011 | SKF |
| Drive End | Ball | .014 | .011 | SKF |
| Drive End | Ball | .021 | .011 | SKF |
| Drive End | Ball | .028 | .150 | NTN |
| Fan End | Inner Raceway | .007 | .011 | SKF |
| Fan End | Inner Raceway | .014 | .011 | SKF |
| Fan End | Inner Raceway | .021 | .011 | SKF |
| Fan End | Outer Raceway | .007 | .011 | SKF |
| Fan End | Outer Raceway | .014 | .011 | SKF |
| Fan End | Outer Raceway | .021 | .011 | SKF |
| Fan End | Ball | .007 | .011 | SKF |
| Fan End | Ball | .014 | .011 | SKF |
| Fan End | Ball | .021 | .011 | SKF |

(All dimensions in inches)

Table 7. [III] Fault specifications of the defected bearings used in CWRU’s tests.

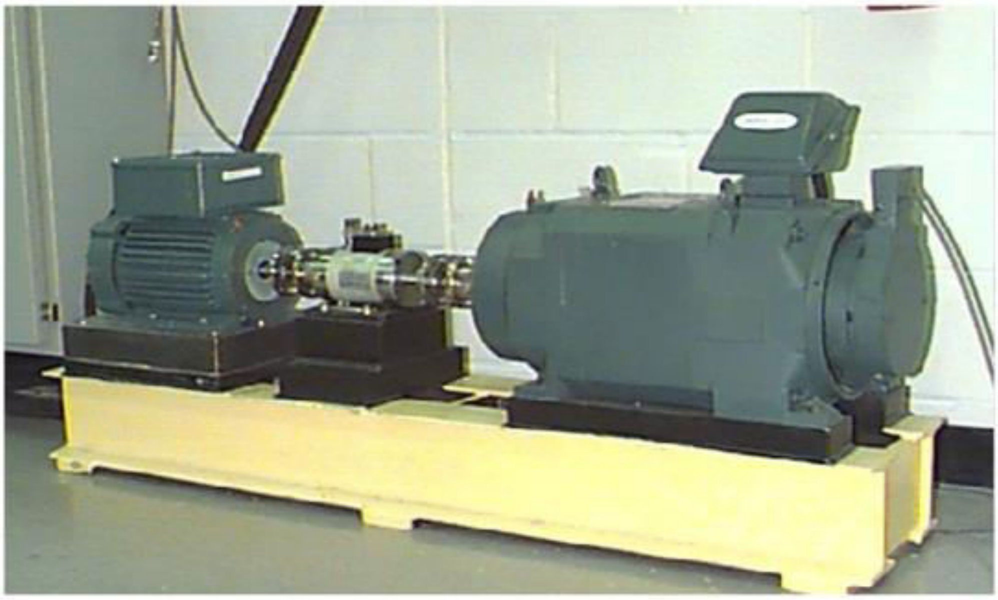


Image 14 [III] The testing rig used by CWRU to construct their publicly available faulty rolling bearings dataset.

As a final note, one can observe in tables 3-5 three different possible positions for the outer race fault to be. For purposes of simplification from now on, as outer faults in the context of this work, only the centered @6:00 outer race faults will be taken into consideration, since the fault frequency remains the same for either of the three possible outer location [III].

## 4.4 Differentiations Between ESFD and Intelligent Methods

Traditional fault diagnosis techniques are pretty straight forward, but they work in an entirely different way than intelligent fault diagnosis. When EFSD is used to determine if a rolling bearing is defected or not, the program expects a big volume of acceleration data, then transforms them from the time domain to the frequency domain, as explained in chapter 2, and in this big volume of frequency domain data, the characteristic fault frequencies and their harmonics are searched for. The program then returns the state-class of the bearing, according to the existence or not of the aforementioned frequencies on the frequency domain representation of its acceleration data. In order to reliably perform the process described above, a significant amount of data is need. In contrast to that when a machine learning algorithm is used, the data are split up to smaller chunks called data samples. Typically, as sample consists of a few hundred data points and each sample is reviewed separately from an already trained machine learning algorithm and classified accordingly. The problem this reality introduces, is that because of the very different way the two methods treat their fed data, they are not directly comparable. The machine learning approaches will be evaluated in a percentage of successful data sample classification, while the ESFD approach can only be evaluated on the grounds of correct classification of the entire volume of data samples, as a whole. So to conclude on that, while the machine learning approaches will be compared to each other on a percentage of success basis, they will be compared to the ESFD method on different merits like computational efficiency and their ability for real time application.

## 4.5 ESFD Implementation on CWRU's Data

To implement the ESFD method, the very first thing that needs to be done is calculating the location specific fault frequencies for the rolling bearing under study. Following the information from table 6 and using the equations [4.1] – [4.3] the fault frequencies for the rolling bearing used in the measurements can be calculated. Also, the fault frequencies can be calculated directly from the table 6's *defect frequencies* section. The multipliers given by CWRU in that section are just the implementation of the relations [4.1] – [4.3]. Using either approach, the following table will result.

| - | Rotation Speed of the Shaft (Hz) | Fault Frequency Multiplier | Fault Frequency (Hz) |
|---|---|---|---|
| Inner Race Fault | 29.93 | 5.4152 | 162.077 |
| Ball Fault | 29.93 | 4.7135 | 141.075 |
| Outer Race Fault | 29.93 | 3.5848 | 107.293 |

Table 8. Location Specific Fault Frequencies (in red).

Using code developed in MATLAB's programming environment, the acceleration data obtained by CWRU, specifically the data for drive end bearings operating under 0hp motor load, 1796 rpm (or 29.93Hz) shaft rotational speed, 12kHz sampling rate and faults with variant diameters, ranging from 0.007 inches to 0.021 inches, are processed and the results are the following. The code used for that can be found in the appendix. Figures 14 – 16, are the ESFD results for a fault of 0.007" diameter. Figures 17 – 19, are the ESFD results for a fault of 0.014" diameter. Figures 20 – 22, are the ESFD results for a fault of 0.021" diameter. Figures 23 – 25, are the ESFD results for a normal bearing against the fault frequencies of the three faults.

As it can be observed from figures 14 – 25, the method generally works. The fault frequencies and their harmonics line up almost perfectly with the highest peaks on their immediate frequency areas which is a great indication that the rolling bearing has that location specific fault. Additionally, in figures 23-25 that a normal bearing is being tested against all three of the fault specific frequencies from table 8, these frequencies or their harmonics do not coincide with any peaks in a statistically important manner, so it is safe to assume that their figures represent a bearing without any of the main faults presented on this work. Unfortunately though, this method is not foolproof, which can become apparent when observing figures 18 and 19. Even though there is a pattern of matching fault frequencies and high peaks to

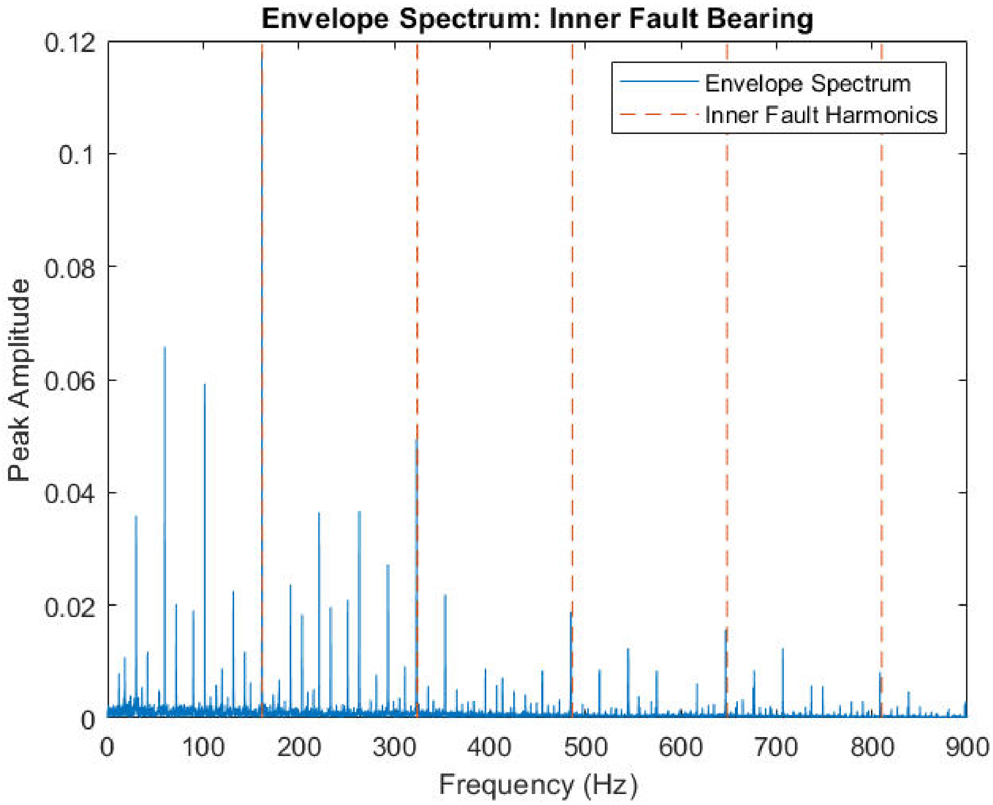


Figure 14. EFSD results for Inner Fault Bearing with a fault diameter of 0.007’’.

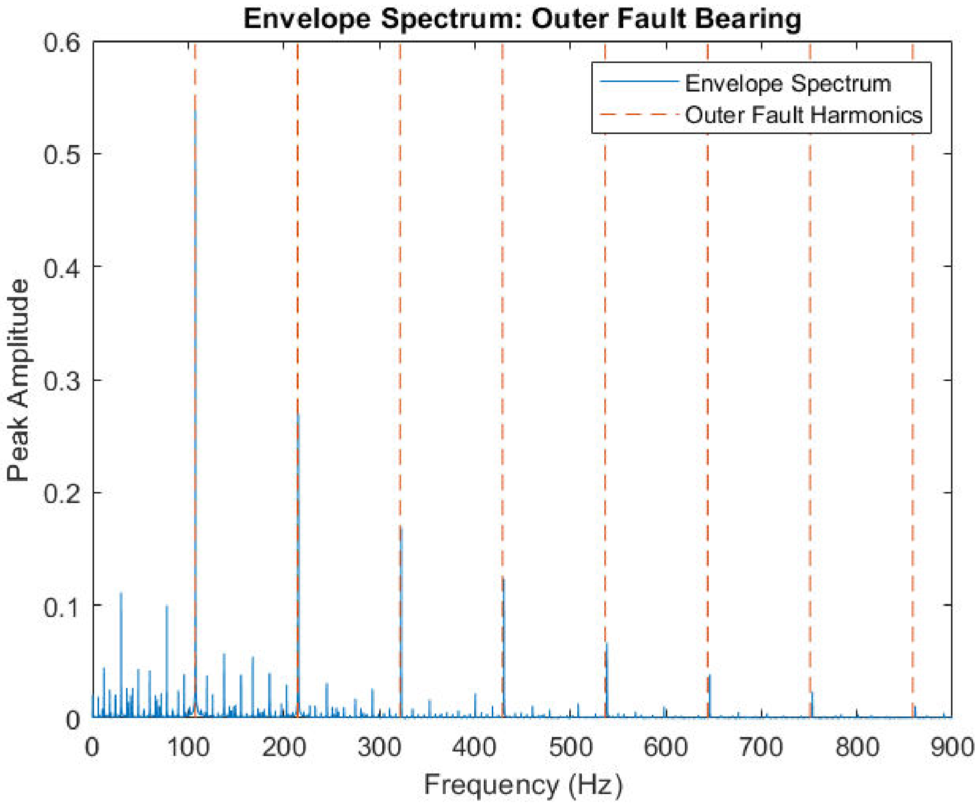


Figure 15. EFSD results for Outer Fault Bearing with a fault diameter of 0.007’’.

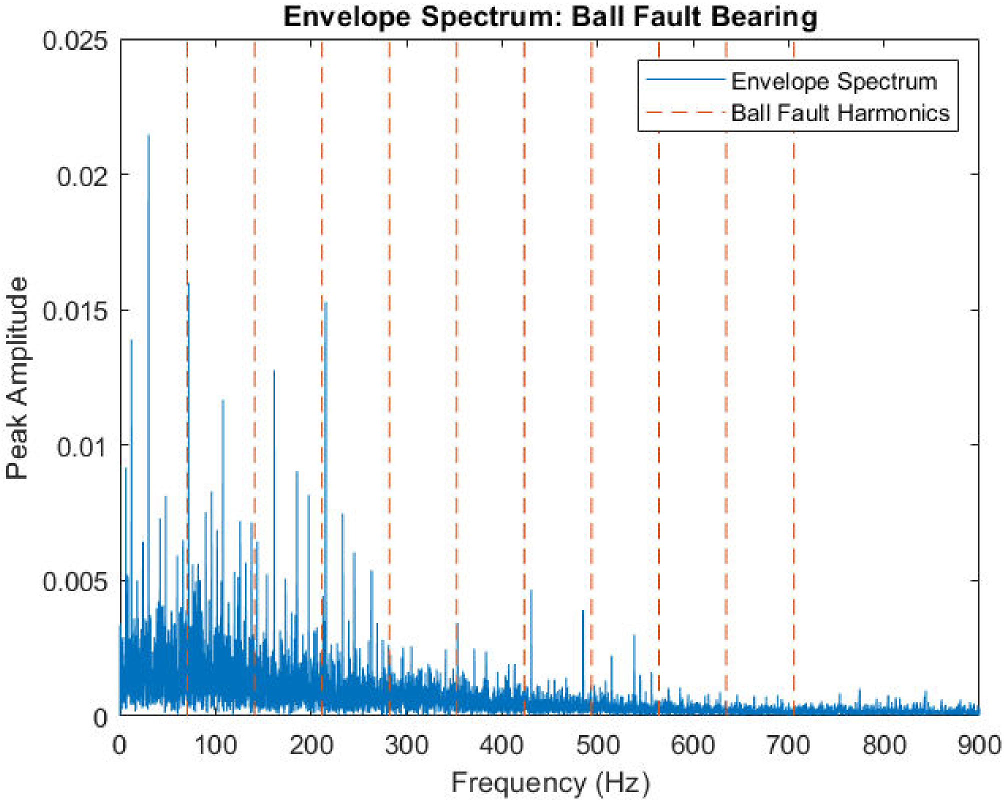


Figure 16. EFSD results for Ball Fault Bearing with a fault diameter of 0.007''.

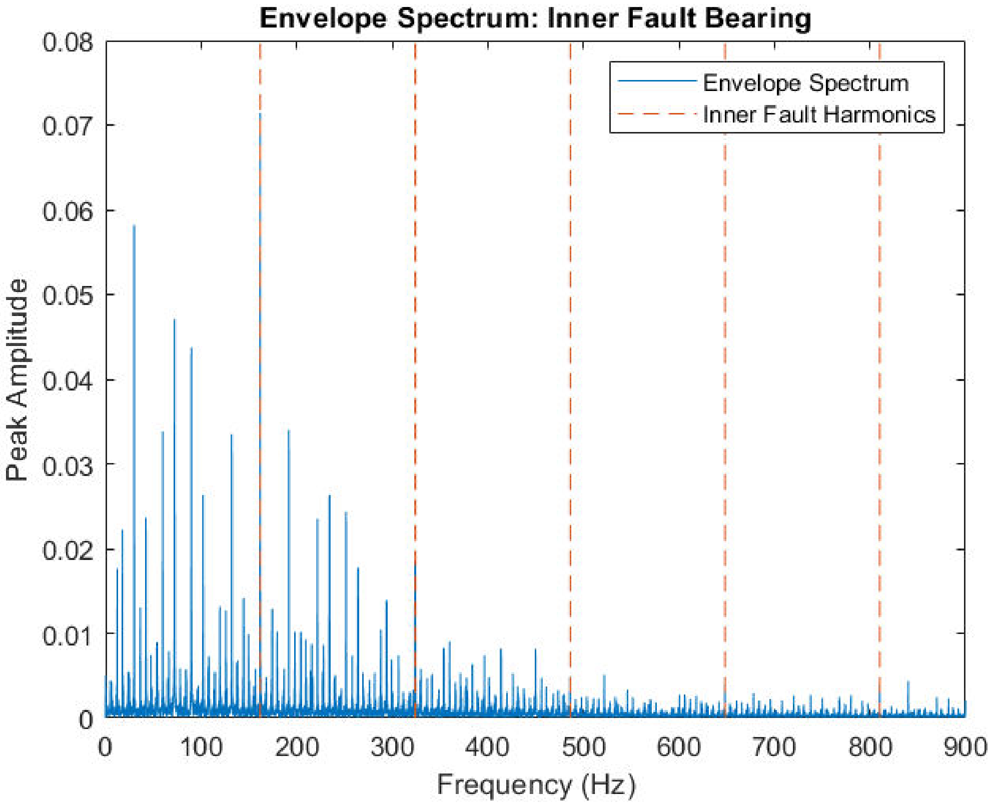


Figure 17. EFSD results for Inner Fault Bearing with a fault diameter of 0.014''.

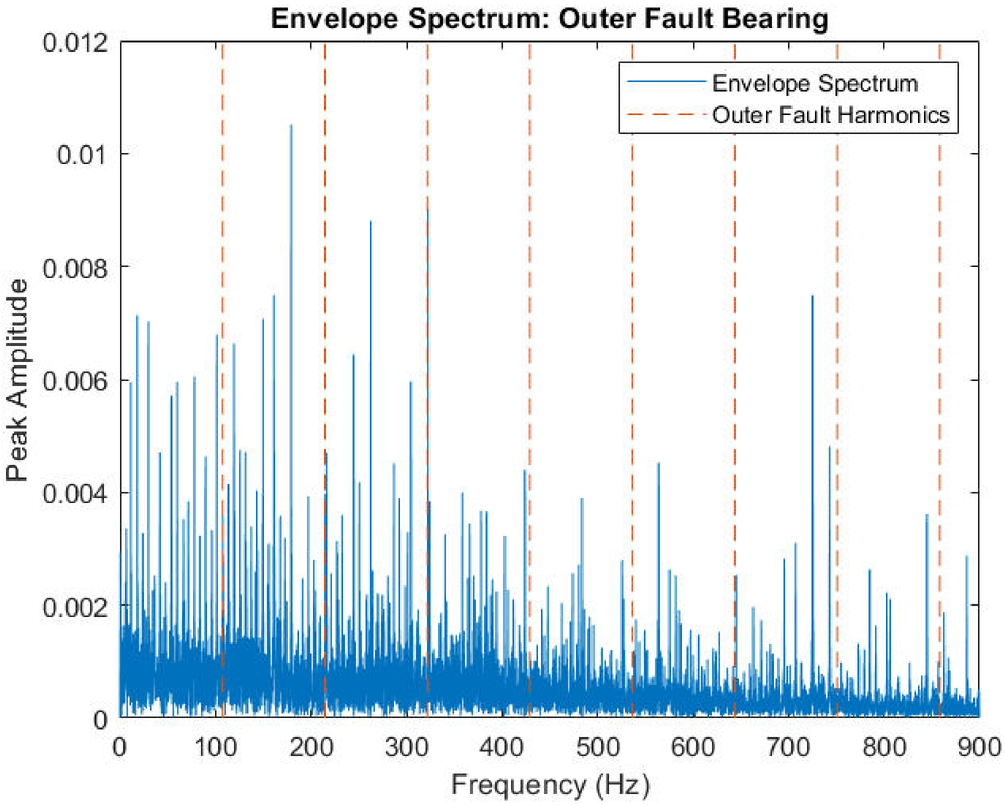


Figure 18. EFSD results for Outer Fault Bearing with a fault diameter of 0.014’’.

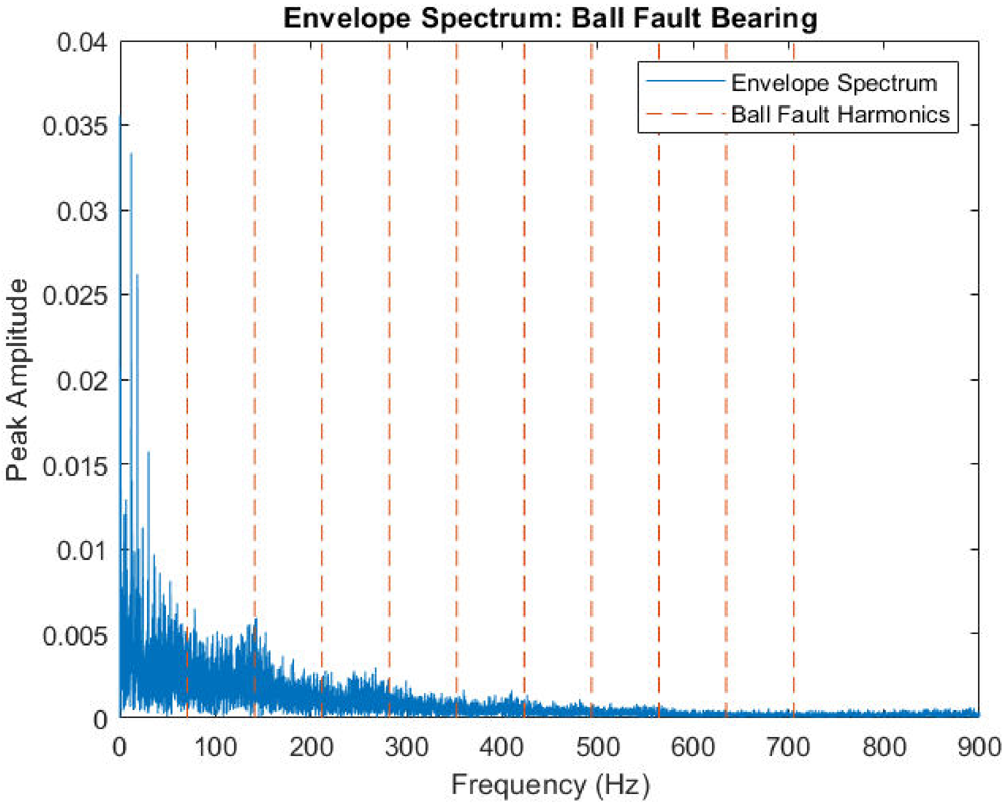


Figure 19. EFSD results for Ball Fault Bearing with a fault diameter of 0.014’’.

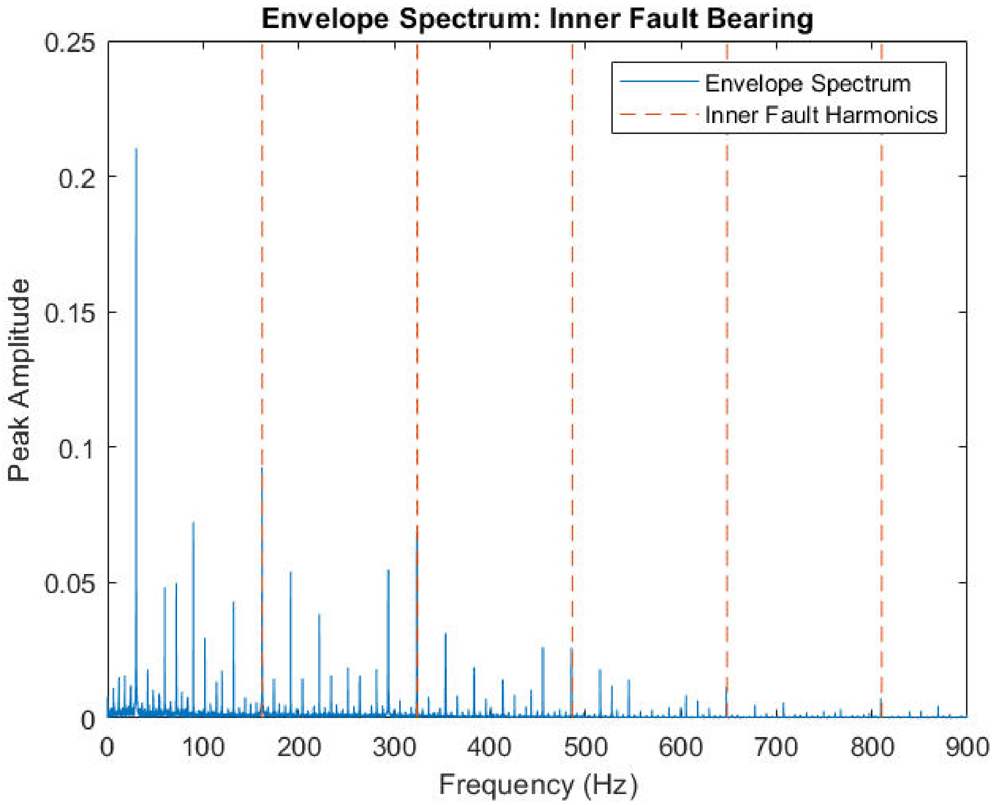


Figure 20. EFSD results for Inner Fault Bearing with a fault diameter of 0.021’’.

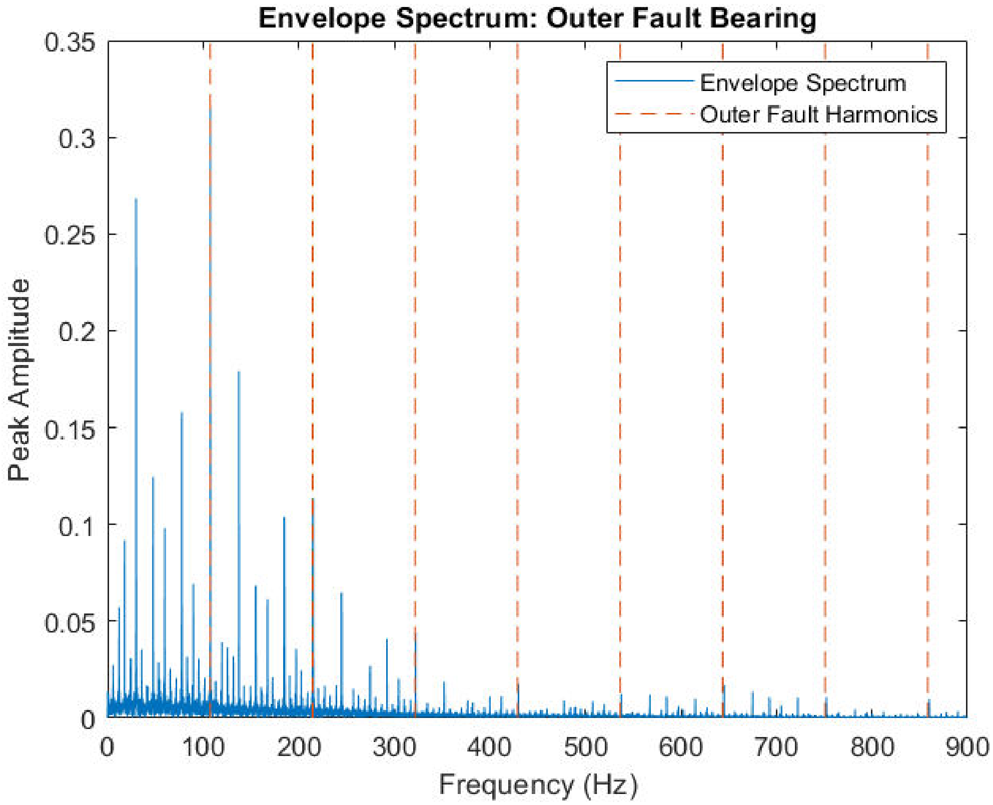


Figure 21. EFSD results for Outer Fault Bearing with a fault diameter of 0.021’’.

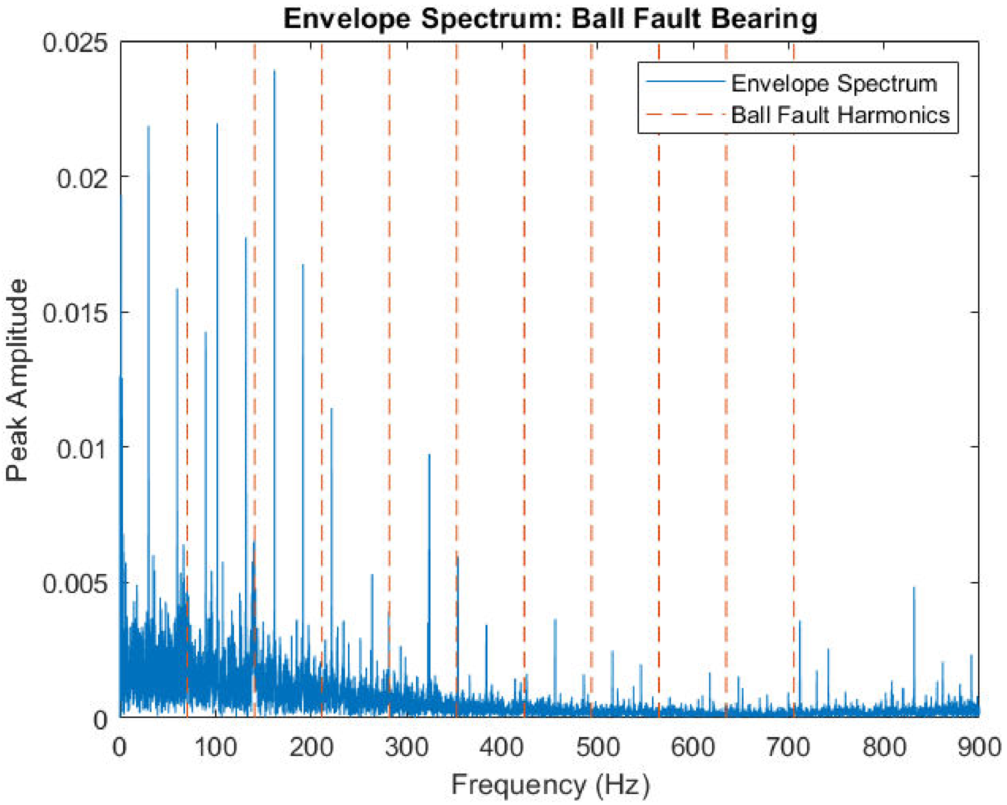


Figure 22. EFSD results for Ball Fault Bearing with a fault diameter of 0.021''.

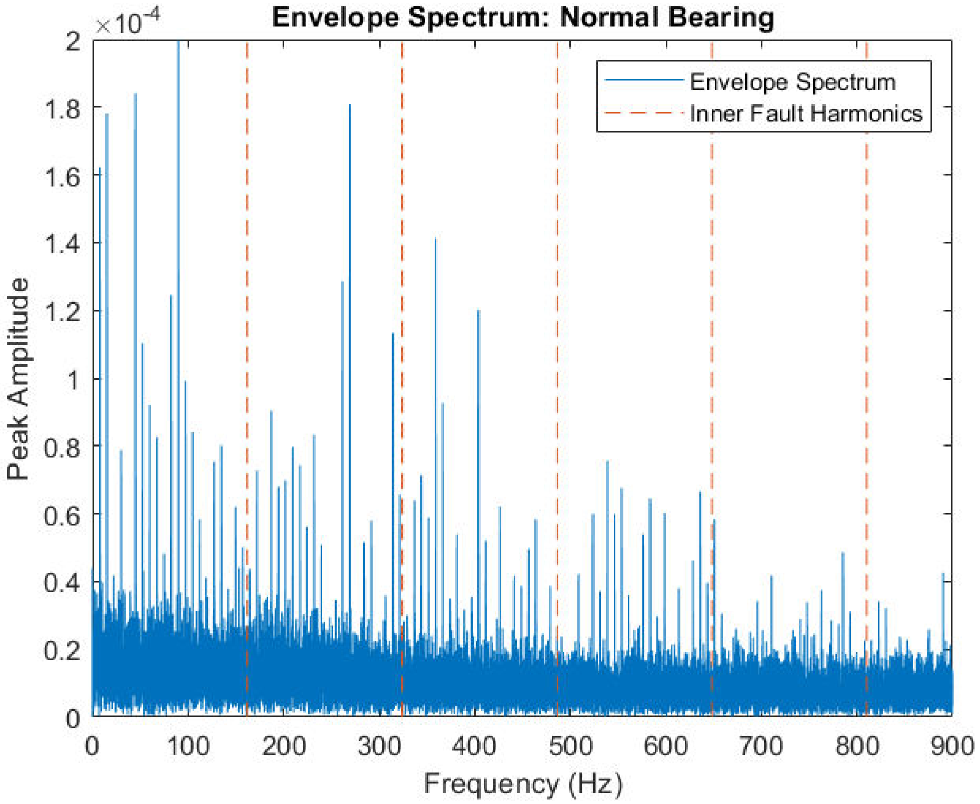


Figure 23. EFSD results for Normal Bearings against the Inner Fault Harmonics.

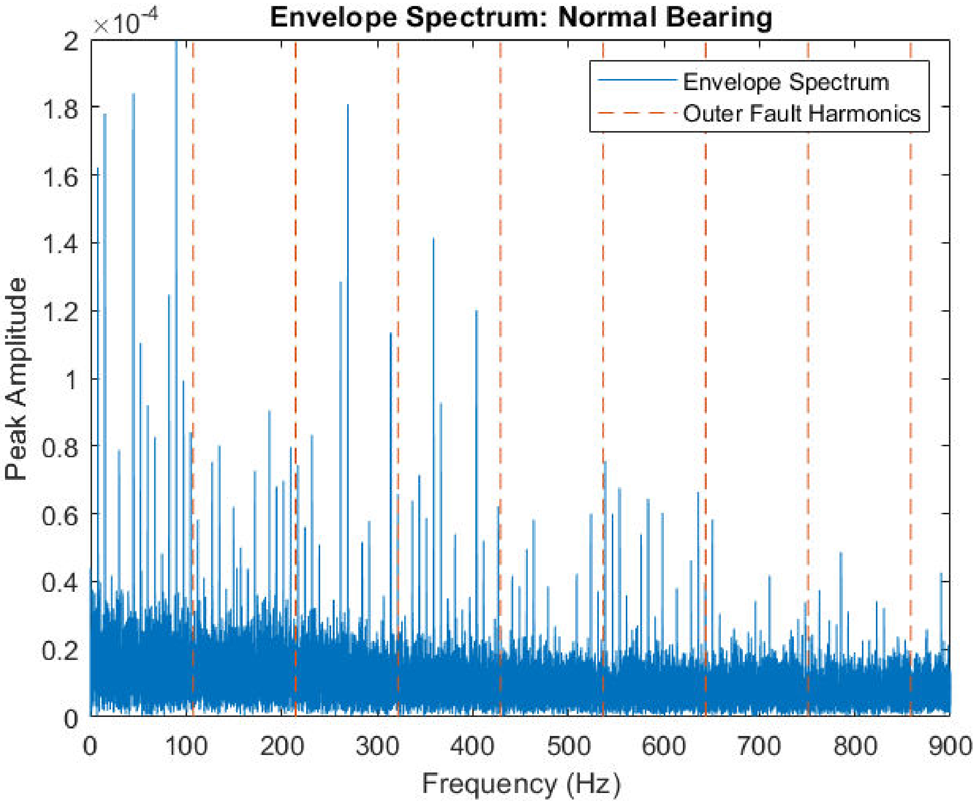


Figure 24. EFSD results for Normal Bearings against the Outer Fault Harmonics.

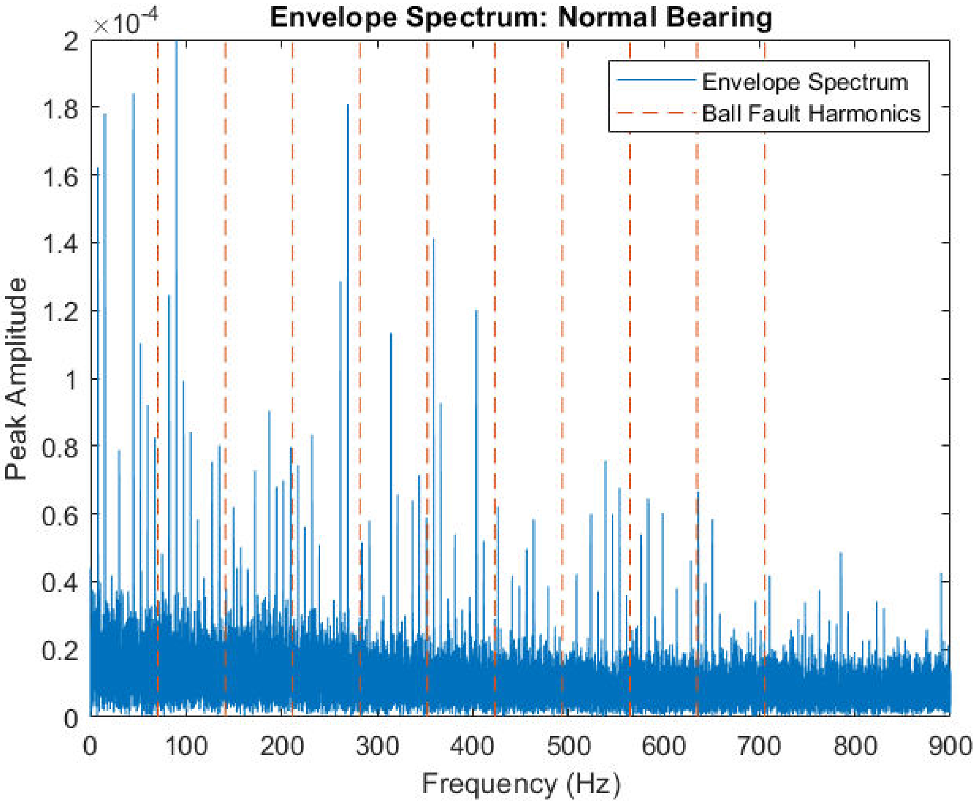


Figure 25. EFSD results for Normal Bearings against the Ball Fault Harmonics.

be found, these figures are much fuzzier than the others and more difficult to definitively claim that they belong to a faulty bearing. The reality of the matter is that, even with the attempts the method makes to eliminate much of the noise, some of it is bound to end up in the final frequency domain representation and make it harder to decide on the fault state of the monitored bearing. That is happening because the frequencies produced by the faults are much smaller than the frequencies from the shaft's rotation, so they are difficult to pin down through the noise.

To conclude, ESFD is a traditional method for bearing fault diagnosis which is used for years in the field with acceptable results overall, but with a few drawbacks that forces researchers on the field to look for alternatives. One of the main drawbacks is the one explained in the previous paragraph. Even beyond that, a serious problem for ESFD is its inability for real time application. To frame that on context, to get the results described in figures 14 – 25, the data samples need were 121,864. Since the sampling rate is 12kHz, these figures are the result of about ten seconds of operation time. Even if the processing time for the implementation of EFSD is to be ignored, these ten seconds are unacceptable for a real time implementation of the method. Additionally, using this approach makes it impossible to know exactly when a fault frequency was detected since the classification is performed from the frequency domain. Everything described above are problems that will be attempted to be solved with the use of intelligent fault diagnosis methods in the remaining of this work.

## 4.6 Intelligent Methods Implementation on CWRU's Data

On this subsection the four aforementioned intelligent methods will be reviewed using four indicative architectures, respectively. These architectures will be sourced from the papers referenced in their respective subsections and they will act as examples of everything presented by now. Before that though, how the data will be split into data samples, training dataset and testing dataset must be addressed.

### 4.6.1 Data Preparation

As mentioned before, there are four possible state-classes any vibrational sample can belong to. These classes are designated with code numbers 0 – 3 as the table 9 clearly represents. The sampling rate of the data used in this work is 12kHz, while the speed with which the shaft is rotating is 1797rpm or 29.93Hz. As a result, each revolution of the shaft corresponds to almost 401 data points of acceleration.

| *Bearing State – Class* | *Corresponding Number Label* |
|---|---|
| Healthy | 0 |

| | |
|---|---|
| Inner Race Fault | 1 |
| Outer Race Fault | 2 |
| Ball Fault | 3 |

Table 9. Classes and their corresponding labels.

The total sum of the data for each class is comprised by 121,000 discrete data points. These data points need to be arranged into data samples in order to be fed into the training pipelines of the intelligent methods. In this work, half a revolution of the shaft is defined as the data sample. In accordance with the description above, each data sample is comprised by 200 data points (which corresponds to 0.01667 seconds of operation), which totals to 605 samples per class. One of the most common rules of thump used for training any machine learning framework is the 70-30 rule, which means that 70% of the dataset will be used for training and the rest 30% for testing. Following that, out of the 605 total samples per class, the 423 will be used for training and 182 for validation and testing. This results to 1,692 data samples which make up a training dataset of 338,400 data points, and 728 data samples which make up a validation dataset of 72,800 data points and a testing dataset of different 72,800 data points. The abondance of data is quite clear, which raises the concern of not efficiently training and testing the Network. To compensate and control for that, a second dataset is used for training and testing in which every sixth data sample is omitted. This results in a total of 212 data samples for training (or 53 per class), 96 for validation (24 per class) and 96 for testing (24 per class) [30] [III].

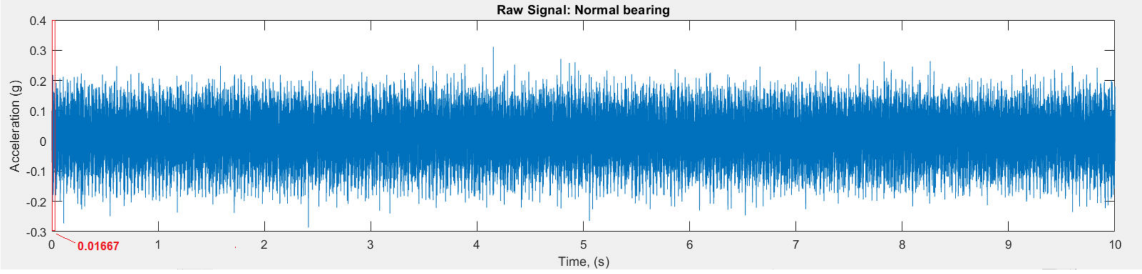


Figure 26. Time domain representation of the Normal Time Series (1st sample in red).

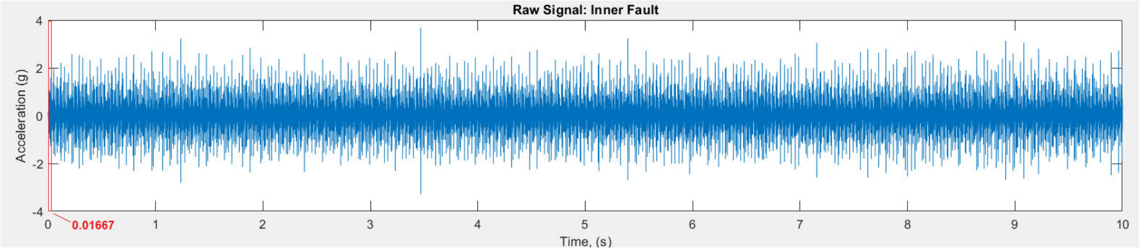


Figure 27. Time domain representation of the Inner Fault Time Series (1st sample in red).

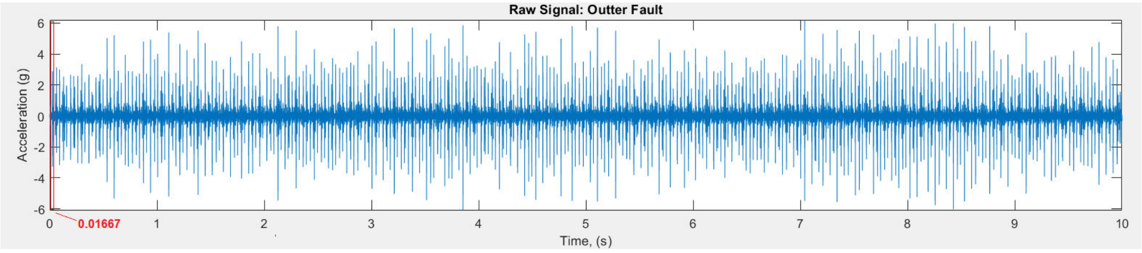


Figure 28. Time domain representation of the Outer Fault Time Series (1st sample in red).

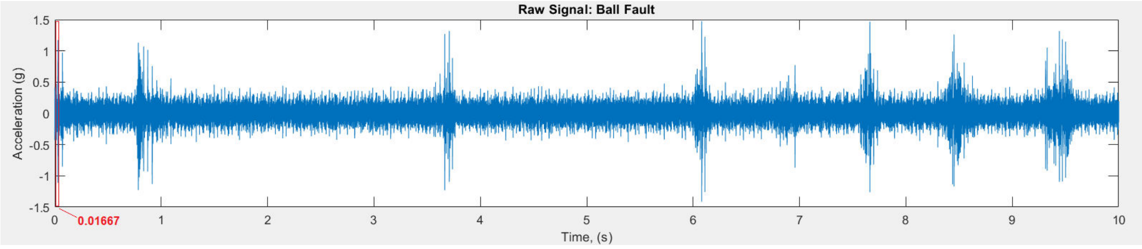


Figure 29. Time domain representation of the Ball Fault Time Series (1st sample in red).

### 4.6.2 MLP Implementation for the CWRU's Data

For the purposes of this work, a three-layer deep MLP will be trained using the backpropagation algorithm. As previously explained, MLPs need their user to define the features of the dataset for them. Handpicking some statics from the time domain or the frequency domain representations of the signal to serve as the data the MLP will use for training, can pose some challenges. For that reason, a powerful signal analysis tool will be used, the *Continuous Wavelet Transform* (CWT). Traditional signal analysis is able to represent an acquired signal to either the time domain or the frequency domain. That reality can be quite limiting, since in many cases temporal information is just as important as frequency information. In the context of this case study for example, the goal is to determine if a data sample (acquired in the short span of half a shaft's revolution) belongs to a faulty or healthy rolling bearing. As previously presented, this cannot be reliable if only the frequency domain data are to be used, so in order to enhance the information that can be extracted from the data, CWT is used to provide some temporal context for the MLP to chew on.

The basic idea behind CWT is taking a *wavelet*, meaning a wave-like signal which is time localized, sliding it over the signal that is being transformed, and evaluating how much the wavelet matches with the signal. Formally, this concept can be represented by the following relation.

$$F(\tau, s) = \frac{1}{\sqrt{|s|}} \int_{-\infty}^{+\infty} f(t) \cdot \psi^* \left( \frac{t-\tau}{s} \right) dt \quad [4.4]$$

Where $F$ is the transformed signal, $f$ is the original signal, $\psi^*$ is the wavelet of choice, $s$ is the scale and $\tau$ is the translation parameter [36].

Some context on the newly introduced parameters from [4.4] is highly needed. As previously explained, the wavelet is a time localized signal which needs to be selected in accordance with its application. There are hundreds of different wavelets to choose from, and some of the most well-known can be found in the figures 26-29. For the purposes of this work, the *morlet* wavelet is chosen.

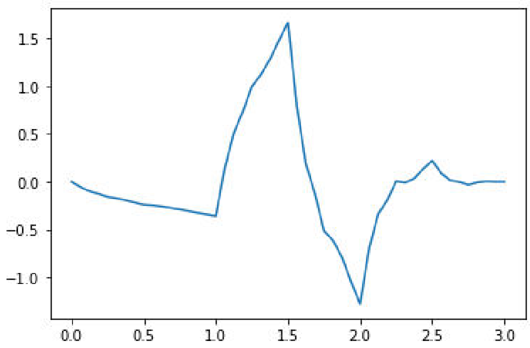


Figure 30. Debauche2 wavelet.

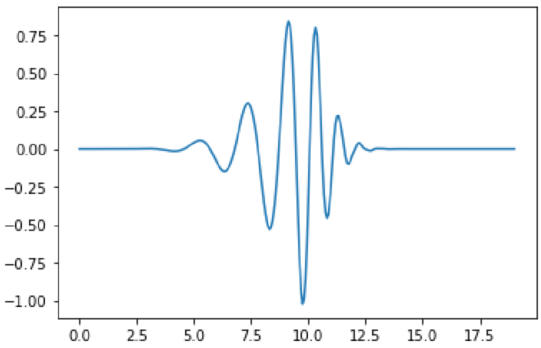


Figure 31. Debauche10 wavelet.

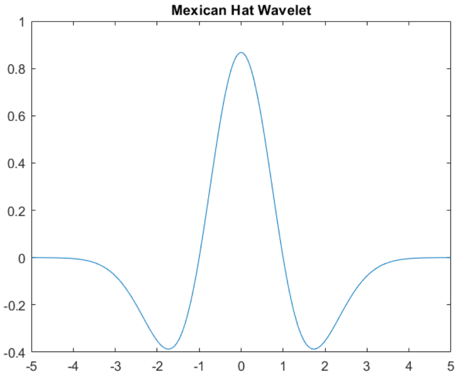


Figure 32. Mexican Hat wavelet.

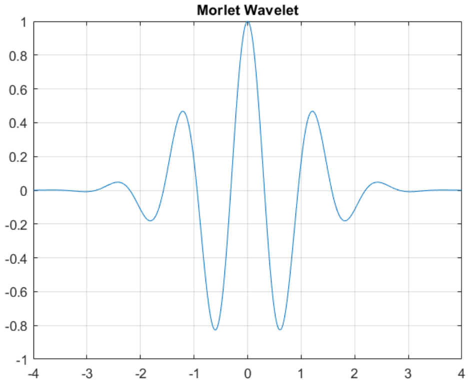


Figure 33. Morlet wavelet.

Additionally, $s$ which stands for scale, is a parameter that controls how spread out in time a wavelet is as the figure 30 clearly represents. Finally, the parameter $\tau$ controls the location of the wavelet and swifts it left to right as figure 31 represents.

Usually, CWT is performed on the signal for multiple scales. Bigger scales typically provide more frequency information, while lower scales provide more temporal information. So, by performing CTW in different scales both temporal and frequency information is extracted from the data [VIII].

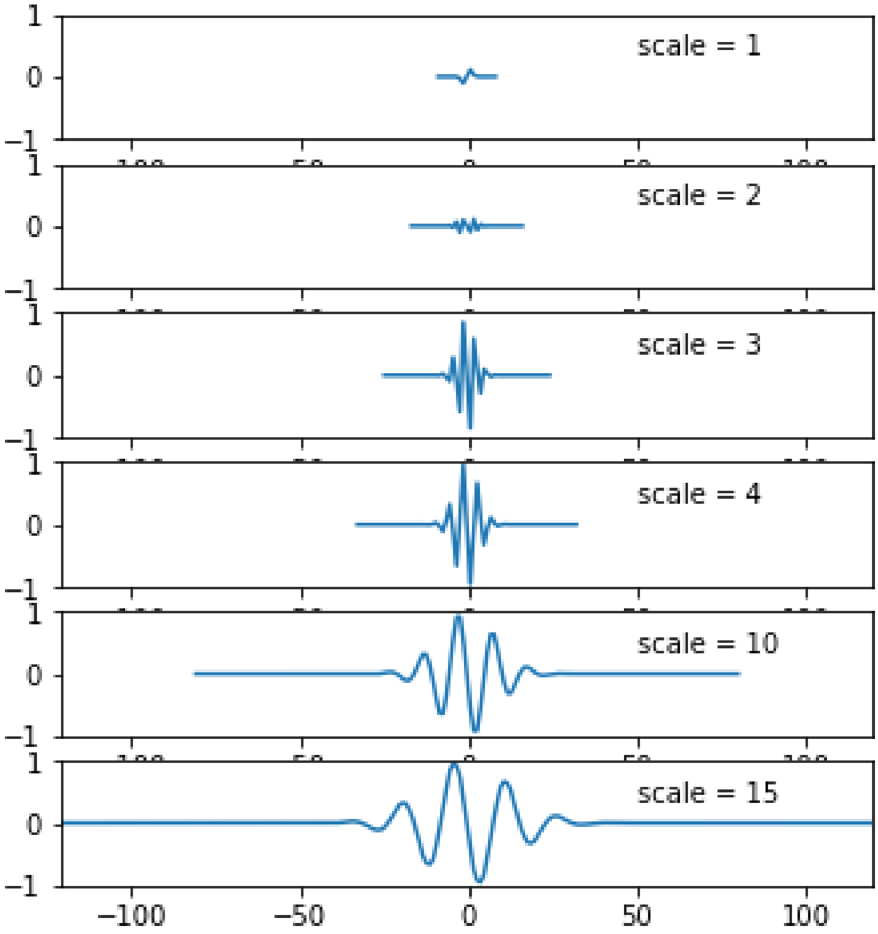


Figure 34. The same wavelet for different scale parameters.

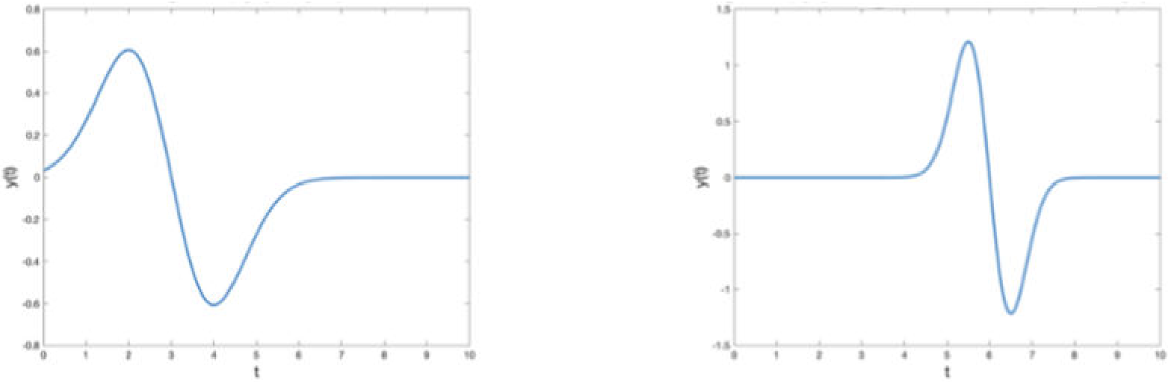


Figure 35. The same wavelet for different time translation parameters.

For the purposes of this work, CWT with a scale range of 1 - 8, a translation parameter of zero and the morlet wavelet will be applied on each data sample. This results to 212 matrixes with dimension 8x200 for shallow training, 8 because of the scale range, and 200 because of the data point's population in each data sample. From each of the scales of the 8x200 matrix, the *root mean square value,* the *crest factor* and the *kurtosis* will be calculated resulting to 24 features for each data sample [36]. To wrap it all together, a final matrix with dimensions 212x24 is constructed. In this matrix each line contains the three statistic values mentioned above for each scale of every data sample of the dataset, so each line encapsulates the information that the MLP will use to train itself. Alongside this matrix, a second matrix with 212x1 dimensions is constructed which contains the labels of every data sample as they were prescribed in table 9. For testing the methods accuracy, the same preprocess is applied to the testing and validation datasets.

The MLP will be comprised from three fully connected layers. These layers will be connected to each other using the tanh activation function, while the final layer will output its results through a sigmoid activation function. The first layer of the MLP will be comprised from 8 nodes, the second from 9 nodes and the third from 8 nodes. The MLP will be trained using the Adam optimizer, the backpropagation algorithm and the cross-entropy loss function. The architecture presented above, is also represented visually in the image 15. This particular architecture converged using a batch size of 100 and in 200 epochs [36].

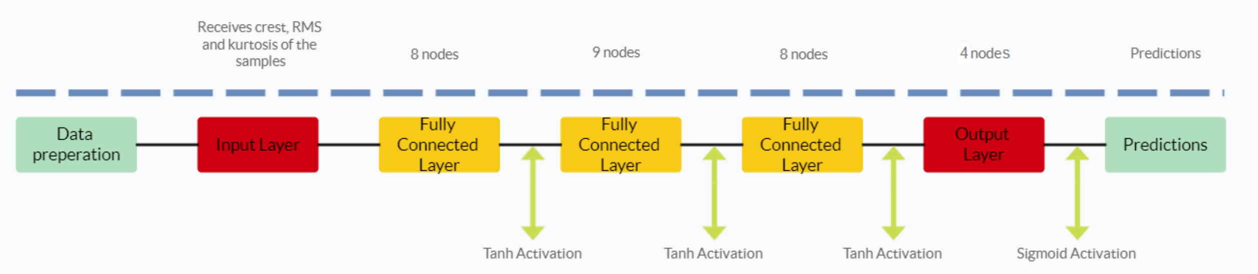


Image 15. Visualization of the MLP architecture.

### 4.6.3 CNN Implementation for the CWRU's Data

For the CNN implementation, there is no feature extraction preprocess. Unlike the more traditional NN such as MLPs, CNNs are able through the convolution that occurs in each layer, to automatically extract the features from any given set of data. One-dimensional CNNs are great for time series feature extraction, because the kernel windows is moving across the time dimension of the data it is being fed, making it very good for time-sensitive datasets (see image 16). Of course, that is a huge advantage for their implementation, since no additional preprocess is needed, and they are ready to accept properly formatted, but raw acceleration data. So, the only thing that needs to be done is setting up the NN and feeding it the training and testing data as they were presented in subsection 4.6.1. Therefore, the input to the Network is a 212x200 matrix with the training data and its corresponding 212x1 targets matrix and a 96x200 matrix with the testing data withs its corresponding 96x1 targets matrix Also, a 96x200 matrix with the validation data and its corresponding 96x1 targets matrix is used during training.

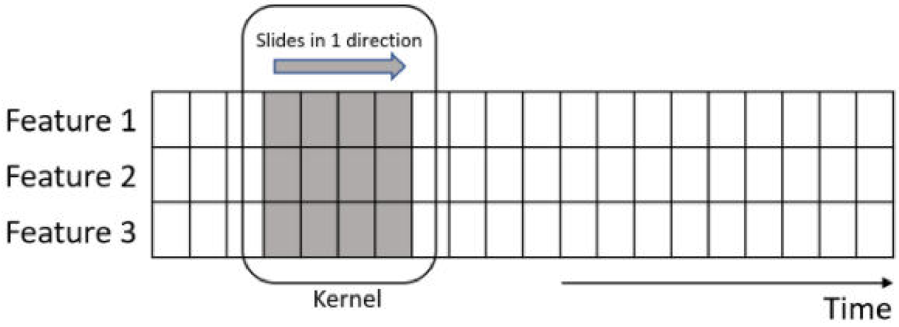


Image 16. [IX] Why an 1D-CNN can be good for time-series date.

The architecture used in this implementation consists of three one-dimensional convolution layers, intercepted by two max-pooling layers and then followed by two fully connected layers to actually perform the classification and output the results. The first convolution layer has 60 nodes, the second 40 and the third also 40, while the fully connected layer consists by 20 layers. The filter size of the convolution layers is set to 9, stride is set to 1 and the padding is set to 0 for all three layers. For the pooling layers, their dimension is set to 4 and their stride also to 1 for both layers. The activation function tanh is selected across the board for all the layers (expect the pooling layers which do not need one), and the output of the layer is filtered through a SoftMax activation function. Also, to ensure that overfitting will be avoided, the maximum number of epochs is set to 20 and if the classification accuracy of an epoch reaches 98%, training is automatically terminated [35] [37]. To train the CNN, the Adam optimizer, the backpropagation algorithm and the cross-entropy loss function will be used. A visual representation of the architecture can be seen in image 17.

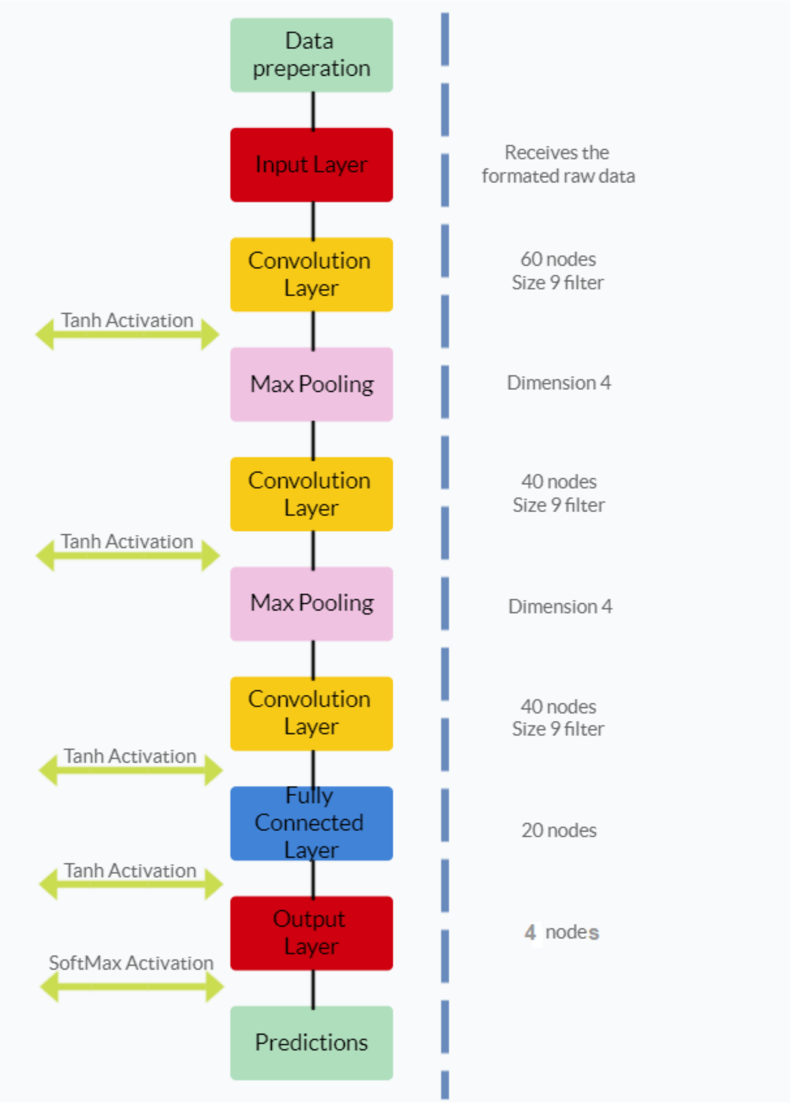


Image 17. Visualization of the CNN architecture.

### 4.6.4 C-CNN Implementation for the CWRU's Data

CNNs are a great option for a big variety of classification problems, especially for time series and image classification problems, due to their ability to extract features from the data on their own. One big problem with CNNs though, is that their effectiveness is highly depended on their filter size. The usual approach to deal with that problem is to perform multiple experiments with different filter sizes and selecting the optimal. In order to tackle that problem an implementation named *Concurrent Convolutional Neural Network* (C-CNN) was introduce by [38]. C-CNN utilizes a parallel multi-branched architecture, in which the raw vibrational data are being simultaneously input in five iterations of the same architecture, but each iteration has a different scale of filter sizes on the convolutional layers. The output of each branch is fed into a *concatenation* layer in which the extracted features of each iteration are fused together. Then the concatenated data are passed through a *flatten* layer which properly formats them to be fed into the fully connected layer which will perform the final classification. A visual representation of the C-CCN architecture can be seen in image 17. Another biproduct of that architecture, other than eliminating the need for selection of just one filter size arrangement, is that because the features are extracted from many different filters, they describe the data with more nuisance and detail than a normal CNN could. The input to the Network is for shallow training is a 212x200 matrix with the training data with its corresponding 212x1 targets matrix, a 96x200 matrix with the testing data withs its corresponding 92x1 targets matrix and a 96x200 matrix with the validation data and its corresponding 92x1 targets matrix.

Each branch consists of two convolutional layers and two pooling layers. Every convolutional layer has 64 nodes, 0 padding and a stride of one. Each convolution is performed with filter of size 5, 25, 50, and 125 in both layers of each branch. The pooling layers have a dimension of 10, with 0 padding and stride of 1. Finally, the fully connected layer has as many nodes as the classes of the problem. The activation function transferring the output of the convolutional layers is ReLU in every case, and the fully connected layer outputs its predictions through a SoftMax activation function. To train the C-CNN, the Adam optimizer, the backpropagation algorithm and the cross-entropy loss function will be used. This particular architecture converged using a batch size of 150 and in 20 epochs.

### 4.6.5 SVM Implementation for the CWRU's data.

Just like the MLP before it, the SVM implementation needs to be given a selection of features that can describe the signal in order to use them for its training. So, once again the CWT is being deployed to help with that problem. Same as before, CWT with a scale range of 1 - 8, a translation parameter of zero and the morlet wavelet will be applied on each data sample to preserve some consistency between the experiments.

For the purposes of this work, a nonlinear SVM with a Radial Basis Function (RBF) kernel is employed. The definition of RBF can be seen in [4.5].

$$K(x,y) = \exp\left(-\frac{\|x-y\|^2}{2\sigma^2}\right) \quad [4.5]$$

Where $K$ is the kernel as presented in subsection 3.8, $x$ are the data points of the training dataset, $y$ is the label of the data point and $\sigma$ is the variance of the data [9] [10] [11].

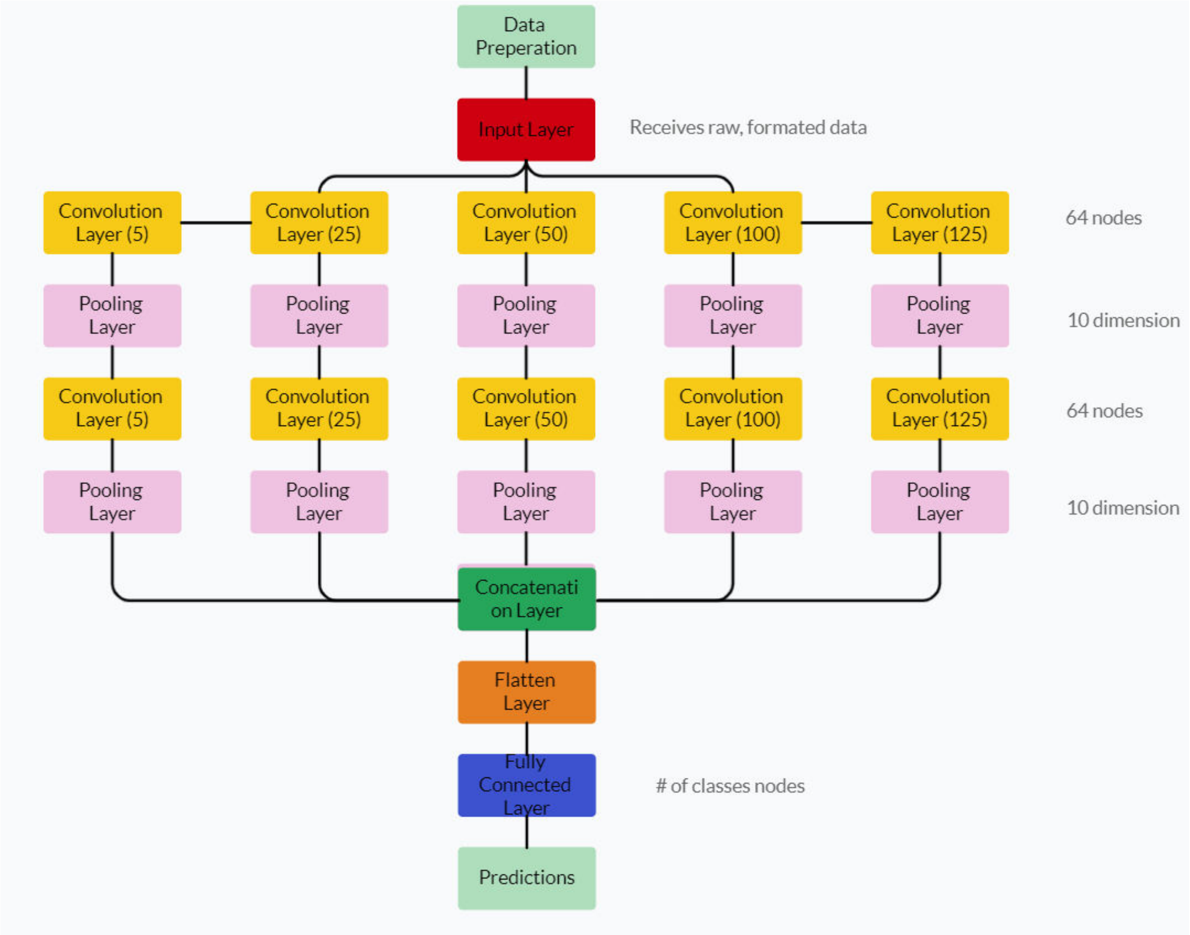


Image 18. Visualization of C-CNN architecture, in the parenthesis of each convolutional layer the filter size of each layer can be seen.

## 4.7 Results and Method Comparison

The appraisal of each intelligent method will be performed in many different levels. First, for the testing dataset a classification accuracy metric is calculated, which represents how many out of a hundred data samples were classified correctly. In addition, the precision metric is calculated, which represents the percentage of the true positive classifications in each class. Also, the recall metric is evaluated which represents the percentage of the actually true positive samples that were classified as true positives. The final metric will be the so-called *f1 score*, which is the harmonic mean of the precision and recall metrics. A good f1 score, suggest that the model balances precision and recall effectively. Finally, a *confusion matrix* for the testing dataset is contracted. Confusion matrixes are a popular way of reviewing the classification results of any classification method. They are matrixes with dimension of $(number\ of\ classes)x(number\ of\ classes)$, the horizontal axes contains the predicted class according to the method, and the vertical axes contain the actual class. That way, one can see how the model classified each sample, and it provides more information about what mistakes the model made. As a final comparison metric, the time each method needed to complete its training is reviewed, alongside the time every model needs to make a prediction for just one sample. Everything above is presented in the following tables.

First, the time every method needed to complete its training can been seen in table 10. The overall training time is the worst case (C-CNN) about 41.5 seconds and in the best (SVM) almost 2.01 seconds. These training times are almost non-consequential because once the training of a model is done, the weights and biases that optimized the model are stored and they are ready to use in any other data sample.

| - | SVM | MLP | CNN | C-CNN |
|---|---|---|---|---|
| Time | 2.01sec | 14.2sec | 4.8sec | 41.5sec |

Table 10. Training time for each model.

In continuum, the time each model needs to make a prediction for just one data sample is presented in table 11. These times are almost 0, which means that every single one of these models is able to be used in a close to real-time application.

| - | SVM | MLP | CNN | C-CNN |
|---|---|---|---|---|
| Time | 0.000sec | 0.068sec | 0.066sec | 0.085sec |

Table 11. Predictions of a data sample for each model and each dataset.

| - | MLP | CNN | C-CNN |
|---|---|---|---|
| Number of Layers | 4 | 7 (Con-Pool-Con-Pool-Con-Fully-Fully) | 4*5 (parallel, Con-Pool-Con-Pool)<br>+<br>3*1 (Concatenation-Flatten-Fully) |
| Number of Nodes-Kernels | (8, 9, 8, 4) | (60, 4, 40, 4, 40, 20, 4) | (64, 10, 64, 10, -, -, 4) |
| Kernel size (for CNNs) | - | (9, 9, 9) | Check image 17 |
| Training time in seconds | 14.2 | 4.8 | 41.5 |

Table 12. Comparative structures of the Neural Networks.

Following that, in tables 13 – 14 the accuracy with which each model predicted the class of the data samples in the final iteration of training and in the testing datasets is presented. It's evident from the table, that all the methods yielded respectable results when tested in a dataset which contained defects with diameter of 0.007'', even if they had never encountered that data before. Especially the convolution-based C-CNN, which returned perfect results. As a result, considerable generalization ability can be achieved, even for samples that are different from the ones used for the training of the models. Additionally, in the tables 15 – 18, the Precision, Recall and F1 scores of each model for the classification of the testing data can be seen.

| Training data | SVM | MLP | CNN | C-CNN |
|---|---|---|---|---|
| Accuracy | 96.87% | 98.96% | 95.83% | 100% |

Table 13. Accuracy metric for the testing data.

| Test Data | SVM | MLP | CNN | C-CNN |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| Accuracy | 96.87% | 97.91% | 95.8% | 100% |

Table 14. Accuracy metric for the validation data.

| SVM - Test | Precision | Recall | F1 |
|---|---|---|---|
| 0 | 100% | 100% | 100% |
| 1 | 96% | 92% | 94% |
| 2 | 92% | 100% | 96% |
| 3 | 100% | 96% | 98% |

Table 15. Different classification metrics for testing data with SVM, organized by class.

| MLP – Test | Precision | Recall | F1 |
|---|---|---|---|
| 0 | 100% | 100% | 100% |
| 1 | 92% | 100% | 96% |
| 2 | 100% | 100% | 100% |
| 3 | 100% | 92% | 96% |

Table 16. Different classification metrics for testing data with MLP, organized by class.

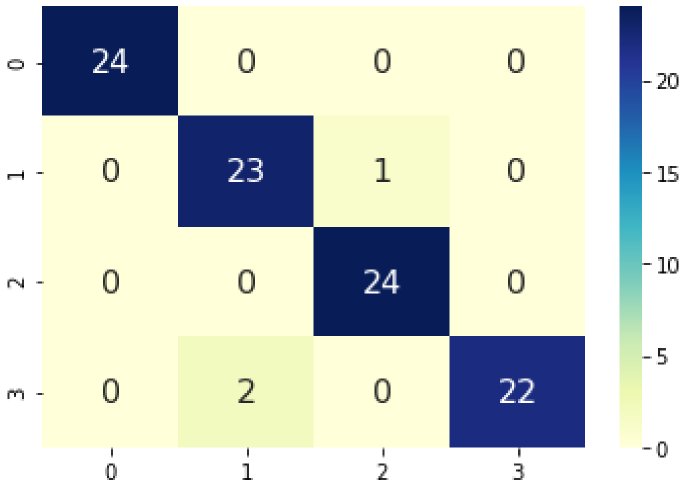
| CNN – Test | Precision | Recall | F1 |
|---|---|---|---|
| 0 | 100% | 100% | 100% |
| 1 | 86% | 100% | 92% |
| 2 | 100% | 83% | 91% |
| 3 | 100% | 100% | 100% |

Table 17. Different classification metrics for testing data with CNN, organized by class.

| C-CNN – Test | Precision | Recall | F1 |
|---|---|---|---|
| 0 | 100% | 100% | 100% |
| 1 | 100% | 100% | 100% |
| 2 | 100% | 100% | 100% |
| 3 | 100% | 100% | 100% |

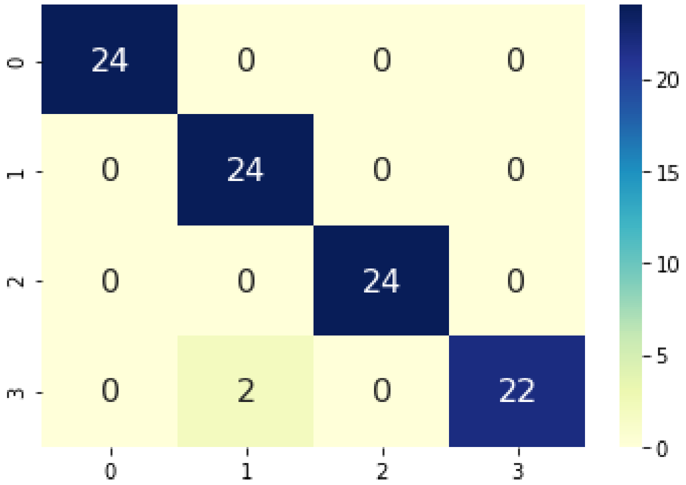
Table 18. Different classification metrics for testing data with C-CNN, organized by class.

To have a better understanding of the way each classification scheme worked, their respective confusion matrixes are presented in tables 19 – 22.



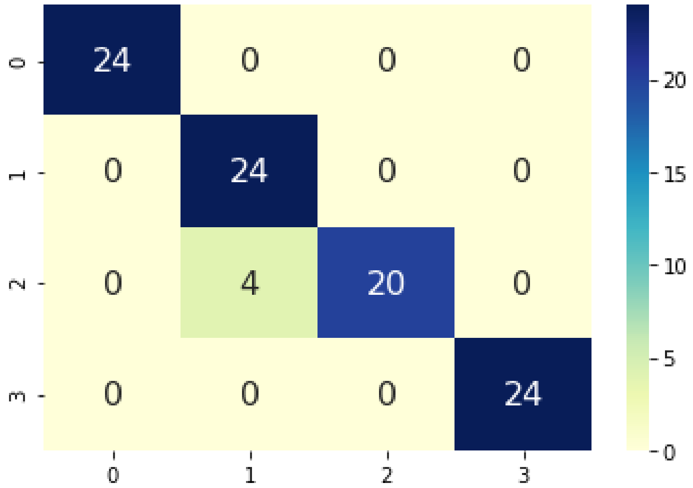
| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 24 | 0 | 0 | 0 |
| 1 | 0 | 23 | 1 | 0 |
| 2 | 0 | 0 | 24 | 0 |
| 3 | 0 | 2 | 0 | 22 |

Table 19. Confusion matrix for the classification of the testing data using the SVM.



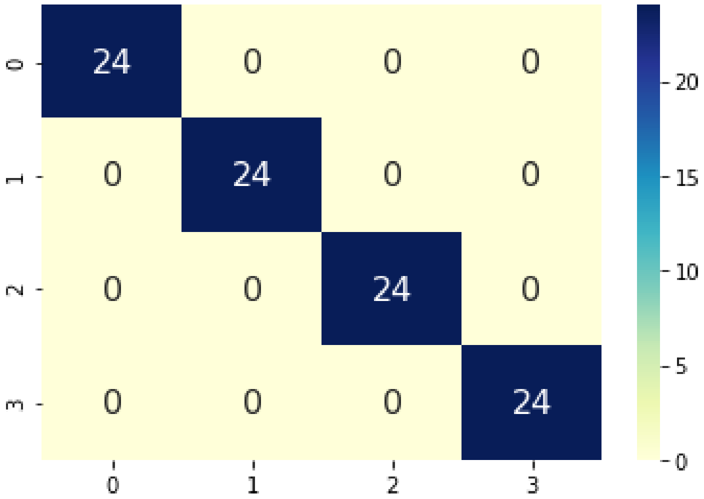
|  | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 24 | 0 | 0 | 0 |
| 1 | 0 | 24 | 0 | 0 |
| 2 | 0 | 0 | 24 | 0 |
| 3 | 0 | 2 | 0 | 22 |

Table 20. Confusion matrix for the classification of the testing data using the MLP.



|  | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 24 | 0 | 0 | 0 |
| 1 | 0 | 24 | 0 | 0 |
| 2 | 0 | 4 | 20 | 0 |
| 3 | 0 | 0 | 0 | 24 |

Table 21. Confusion matrix for the classification of the testing data using the CNN.



|  | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 24 | 0 | 0 | 0 |
| 1 | 0 | 24 | 0 | 0 |
| 2 | 0 | 0 | 24 | 0 |
| 3 | 0 | 0 | 0 | 24 |

Table 22. Confusion matrix for the classification of the testing data using the C-CNN.

Another interesting thing to see is how the value of the loss function converges towards zero over each epoch. These figures can help decide the number of epochs used for training. When the curves tend to became parallel with the x-axis, that is a good number of epochs to choose for training, because the loss of the model will not get any lower than that. Additionaly, it helps to avoid overfitting by minimizing the number of times the model gets exposed to the training set. To that end the following figures are presented in figures 36 – 38 .

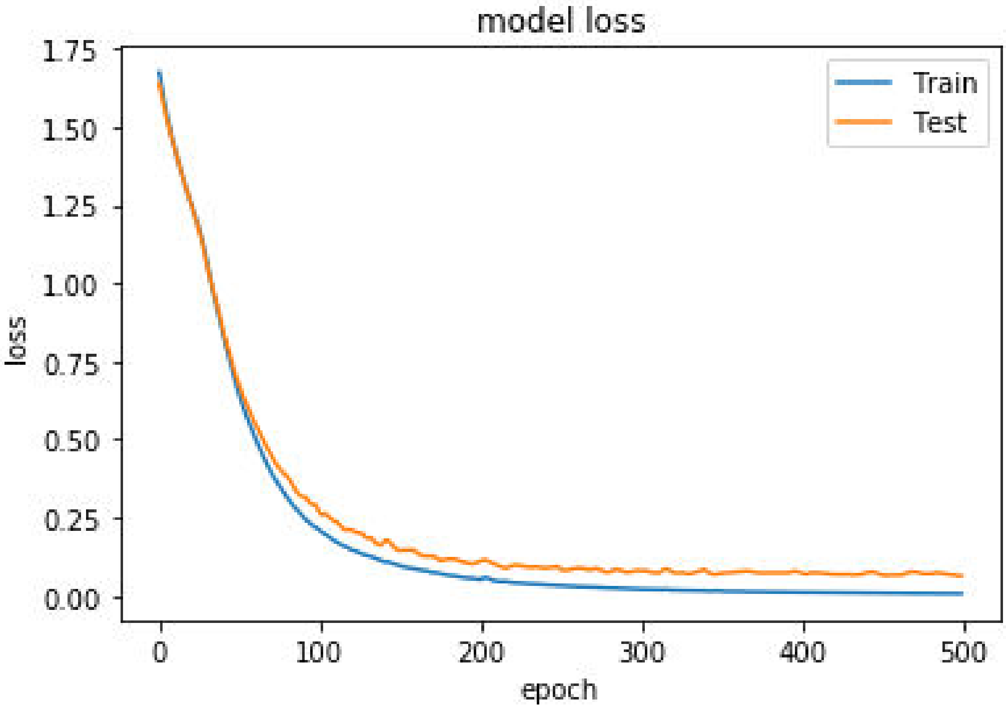


Figure 36. Progress of loss over each epoch for the training of MLP.

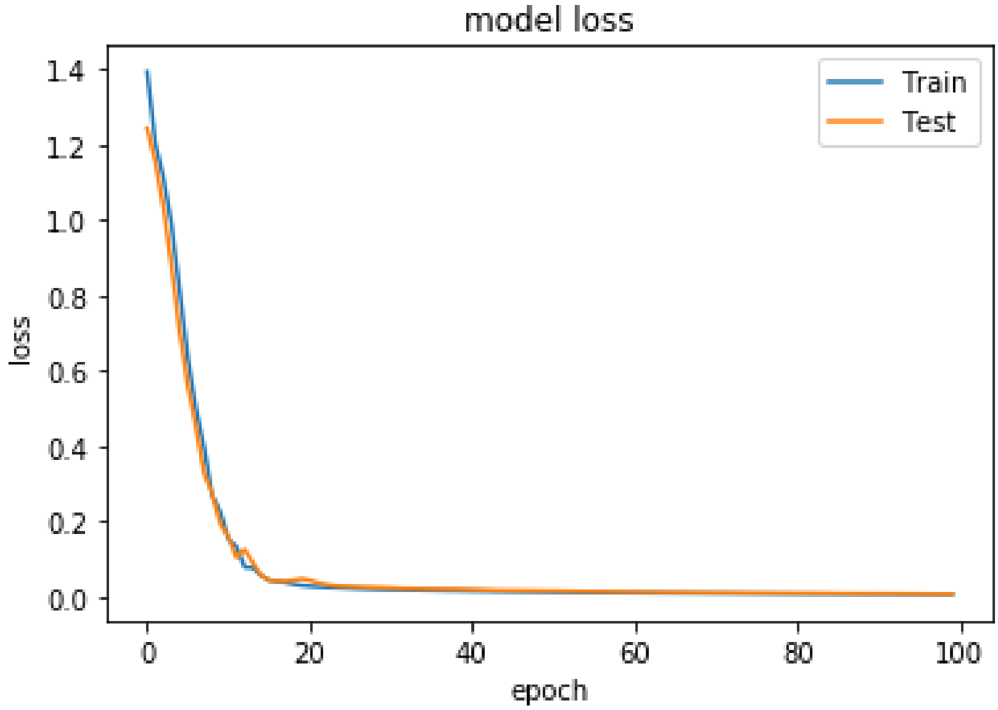


Figure 37. Progress of loss over each epoch for the training of CNN.

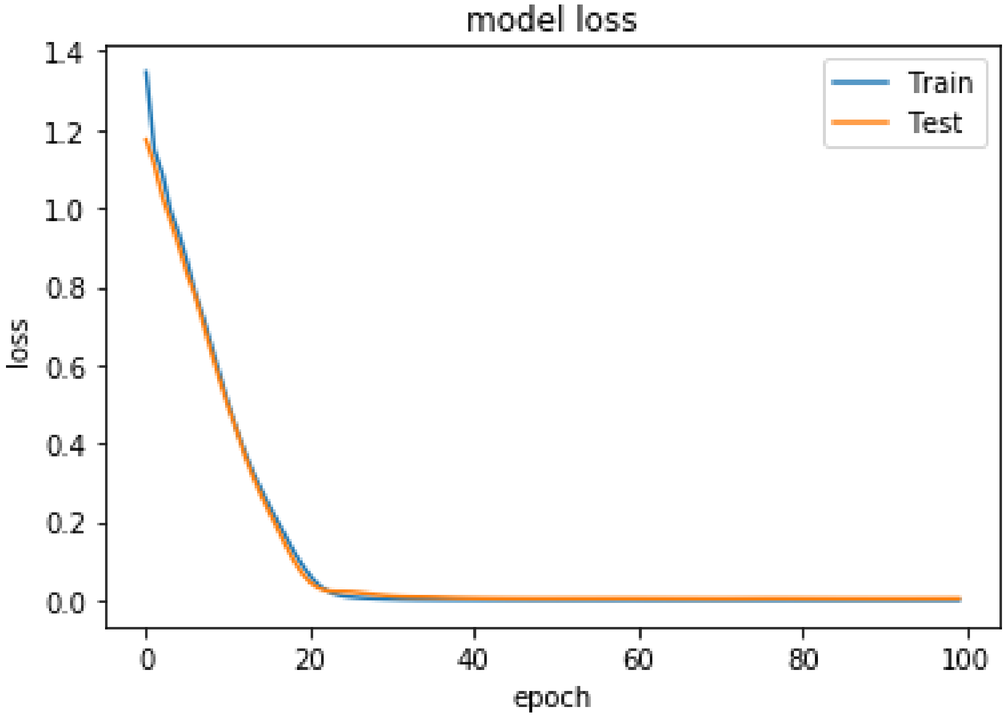


Figure 38. Progress of loss over each epoch for the training of C-CNN.

Figures 36 – 38 ensure that training took place just over enough epochs for the model to converge. If the training were to continue even further the overfitting problem would be making its appearance in the confusion matrices and the metrics of the testing dataset.

Finally, to have a better visual representation of how each model handled the feature extraction and the classification task, the t-SNE [39] dimensional reduction technique is used.

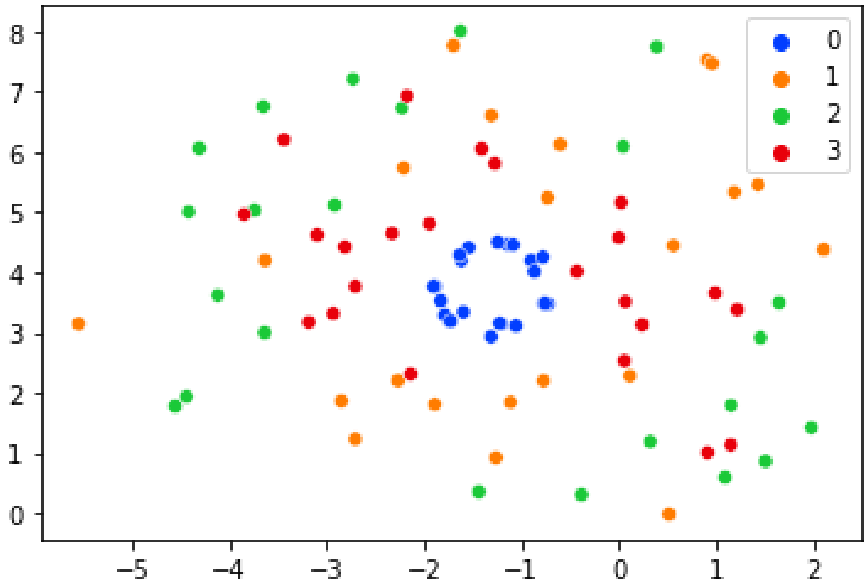


Figure 39. Feature Visualization of the raw data using t-SNE.

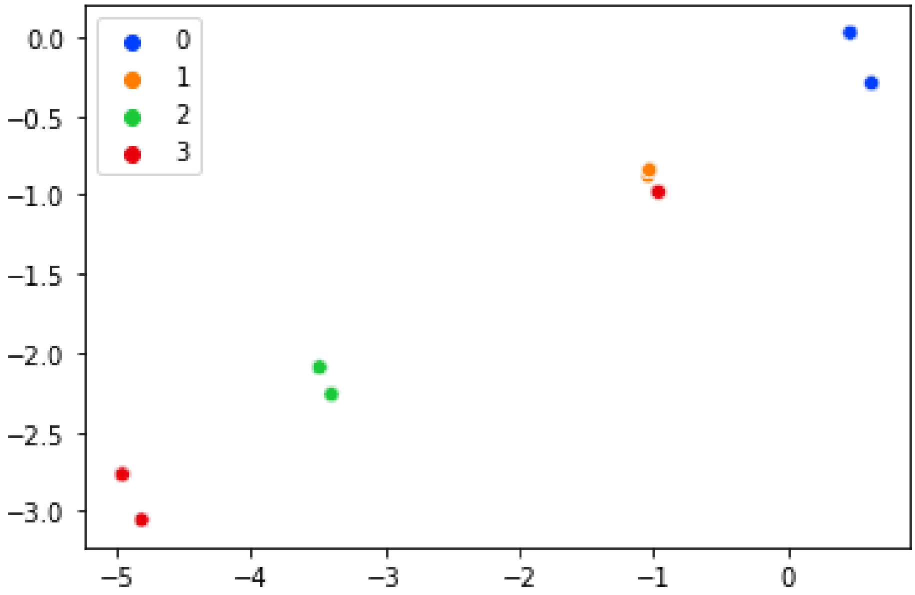


Figure 40. Feature Visualization of the output of the SVM using t-SNE.

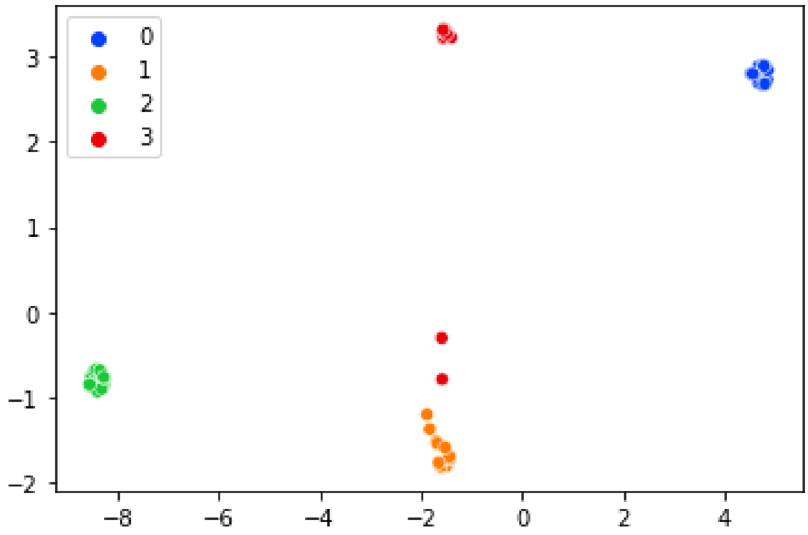


Figure 41. Feature Visualization of the output of the MLP using t-SNE.

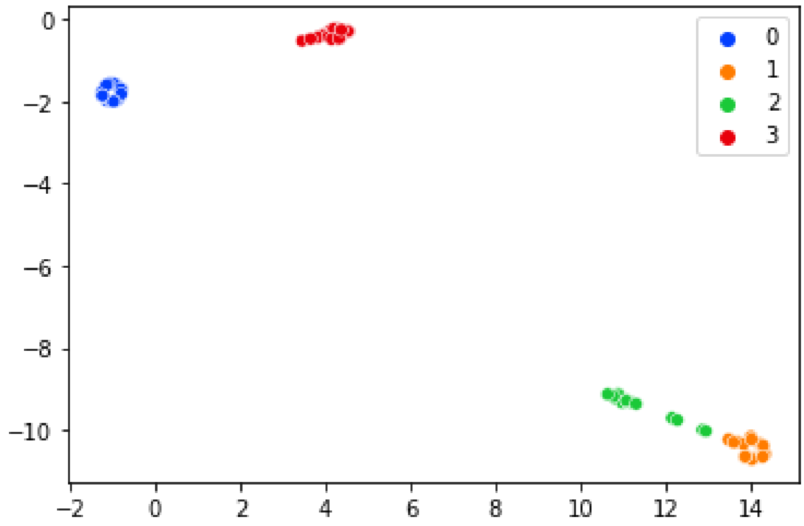


Figure 42. Feature Visualization of the output of the CNN using t-SNE.

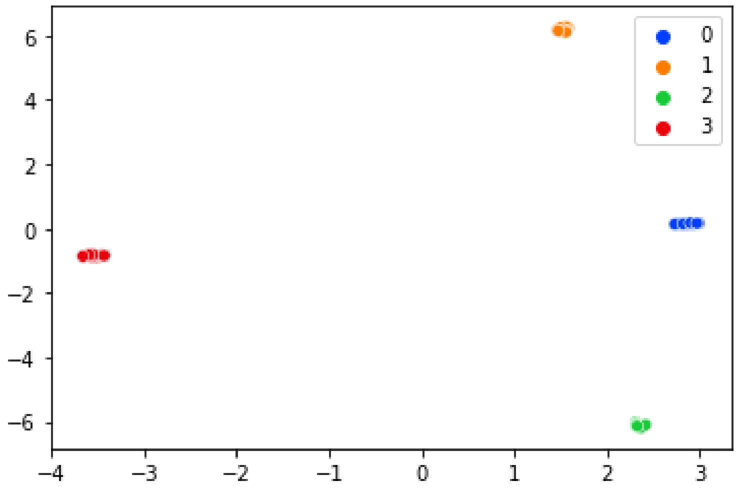


Figure 43. Feature Visualization of the output of the C-CNN using t-SNE.

By reviewing all of the results presented in tables 10 – 22 and in figures 36 – 43, a few conclusions can be drawn. All of the methods managed to successfully classify the testing samples with a very good accuracy, more than 95% in every case. In a sensitive problem like fault detection though, where finding a defective sample could be the difference between system failure and system stability, a good accuracy metric is perhaps not the most important thing to look for. For example, if a healthy sample were to be classified as having an inner fault, or if an inner fault sample were to be classified as a ball fault, the core goal of the application – preventing faulty machinery from keep working – wouldn't be hindered. But if a faulty sample were to be classified as healthy, that could have dangerous side effects for the monitored system. To control for that, the deployed method needs good recall across the board in order to be sure that a high percentage of the true positives of each class is actually found. Also, good precision is needed to make sure that all the samples classified as one class actually belong to that class. In other words, a good F1 score is needed which is a combined metric of recall and precision. Finally, it is fairly important to have a confusion matrix and a t-SNE plot that clearly indicates a low percentage of faulty samples classified as healthy. Even with these much steeper demands, observing the results shows that none of the methods classified a faulty sample as a healthy one and all of the have a F1 score of over 91%, even reaching 100% in multiple cases. So, it is safe to conclude that all of the presented methods work fairly well for the problem in hand. If one needed to be picked out, based on the metrics described above, the C-CNN method would be the clear selection since it scored 100% in all of the used metrics when tested on a never before reviewed part of the data. That being said, the convolutional based methods have another important advantage, they do not need manual feature selection (done with CWT here) like the MLP and the SVM do. That means that raw data, directly from the sensors, can be fed to the model which is always a welcome simplification of the procedure. All of the above lead to the C-CNN being the best of the described methods, even if it is the slowest to trained and the slowest in sample evaluation. It could be important to note that the results obtained in tables 10 – 22 and in figures 36 – 43 does comply with the relevant literature.

Comparing the best intelligent method (C-CNN) with the traditional EFSD method could be challenging. Both methods can provide their user with useful information about the health state of the monitored rolling bearing, but the context the two are working is quite different. If the user needs an online, automated, real-time application that monitors a system continuously, then a well-trained C-CNN model would be the only choice between the two. But if the application requires a system that will periodically check the system and the user will visually inspect the resulted figures, then EFSD will be sufficient. So, the most important factor when considering a choice between traditional and intelligent fault diagnosis techniques, will be the requirements of the end user.

## 4.8 System Specifications

For the implementation of this case study, the computational unit that was used is an Intel CORE i5, 8th Gen CPU. For the EFSD, the MATLAB programming environment was used. For the intelligent methods, the data preparation was performed in MATLAB’s programming environment and the training, testing and validating of all the intelligent methods was performed using the python programming language, version 3.7.10. Also, the Keras deep learning framework for python was used for the compilation, training and validation of the NN reviewed in this work. The SVM implementation was made with the help of the sklearn python module. The CWT needed for MLP and SVM was performed with the help of the pywt python module. Additionally, for the visualization and figures needed in this work, both python and MATLAB were used. Finally, the flow charts present in this work were made with the help of visme. All the code used for this thesis, is present in the Appendix.

# Chapter 5. Conclusions – Suggestions for Further Study

In this thesis, the problem of fault diagnosis on rotating mechanical equipment was presented. The most commonly used traditional method (EFSD) and some of the most common intelligent approaches in the relevant literature (SVM, MLP, CNN) were reviewed. This work had to delve as deep as possible in the theory behind EFSD and Neural Networks, present the main points of interest when a network is devised, review one of the staple datasets in the field of fault diagnosis (CWRU’s). All the above, culminated to a case study that in practice demonstrated everything presented previously and manage to draw some useful conclusions about the applicability of each presented method.

From the analysis that preceded, the following can be concluded. Fault detection can be quite useful in order to ensure maximum performance and safety in any mechanical system with rotating parts. That being said whether EFSD or a C-CNN based approach (since C-CNN was demonstrated to be the best out of the reviewed intelligent methods) is the best choice, depends on each applications needs. Real-time applications are best suited for intelligent methods, while application that are not time sensitive could work just fine with EFSD. Also, the cost of each method needs to be taken into consideration. While EFSD pretty much works out-of-the-box with any application that the fault-specific frequencies are known. In order to implement an intelligent method, big volumes of data needs to be acquired and a detailed study needs to be performed in order to find the best architecture to use in each particular case, which amounts to a bigger deployment cost.

To conclude this work, a few suggestions for further study can be made. One of the areas that the C-CNN and every other intelligent method were demonstrated to lack in, was their generalization ability. The intelligent models were trained using a dataset comprised by faults of 0.007’’ in diameter, in 1797 rpm and with a 0hp motor load. This is a pretty narrow point of view considering that defects can be found on rolling bearing in various diameters, and machines operate in a various rpm and motor loads. A dataset containing data acquired from rolling bearings with more diverse operating motor conditions, and with different diameter of defects to be used for training is one suggestion, but more research is needed. Also, the results of this work could be used to create an online, automated, real-time application to monitor an experimental set-up, in order to evaluate how the concepts of this thesis would work in real operational conditions.

# References

Papers:

[1] Piety, K. R.; Magette, T. E. (1979) Statistical techniques for automating the detection of anomalous performance in rotating machinery

[2] R.B. Randall (1978) Efficient Machine Monitoring using an FFT Analyzer and Calculator

[3] Henrique Dias Machado de Azevedo, Alex Maurício Araújo n, Nadège Bouchonneau (2016) A review of wind turbine bearing condition monitoring: State of the art and challenges

[4] Issam Attoui, Nadir Boutasseta, Nadir Fergani, Brahim Oudjani, Adel Deliou (2015) Vibration-based bearing fault diagnosis by an integrated DWT-FFT approach and an adaptive neuro-fuzzy inference system

[5] Vikram Talekar1, Prof. L. S. Dhamande (2015) Condition Monitoring of Deep Groove Ball Bearing using FFT Analyzer

[6] Amit Aherwar, 2Md. Saifullah Khalid, 3Hemant Kumar Nayak (2012) Vibration analysis of machine fault signature

[7] Levent Eren (2017) Bearing Fault Detection by One-Dimensional Convolutional Neural Networks

[8] Yuanhong Chang, Jinglong Chen, Cheng Qu, Tongyang Pan (2020) Intelligent fault diagnosis of Wind Turbines via a Deep Learning Network Using Parallel Convolution Layers with Multi-Scale Kernels

[9] Qiao Hua, Zhengjia Hea,b, Zhousuo Zhanga, Yanyang Zia (2007) Fault diagnosis of rotating machinery based on improved wavelet package transform and SVMs ensemble

[10] Diego Fernández-Francos , David Martínez-Rego, Oscar Fontenla-Romero, Amparo Alonso-Betanzos (2013) Automatic bearing fault diagnosis based on one-class m-SVM

[11] Junyan Yang, Youyun Zhang, Yongsheng Zhu (2007) Intelligent fault diagnosis of rolling element bearing based on SVMs and fractal dimension

[12] Seokgoo Kim, Dawn An, and Joo-Ho Choi (2020) Diagnostics 101: A Tutorial for Fault Diagnostics of Rolling Element Bearing Using Envelope Analysis in MATLAB

[13] Haidong Shao, Hongkai Jiang, Fuan Wang, Huiwei Zhao (2017) An enhancement deep feature fusion method for rotating machinery fault diagnosis

[14] Rob O'Reilly, Alex Khenkin, and Kieran Harney (2009) Sonic Nirvana: Using MEMS Accelerometers as Acoustic Pickups in Musical Instruments

[15] Viral K. Patel, Maitri N. Patel (2017) Development of Smart Sensing Unit for Vibration Measurement by Embedding Accelerometer with the Arduino Microcontroller

[16] Jing ZHOU, Yong QIN, Linlin KOU, Mitchell YUWONO and Steven SU (2015) Fault detection of rolling bearing based on FFT and classification

[17] Eric Bechhoefer (2018) A quick introduction to bearing envelope analysis

[18] Jianhong Wang, Liyan Qiao, Yongqiang Ye, YangQuan Chen (2017) Fractional Envelope Analysis for Rolling Element Bearing Weak Fault Feature Extraction

[19] Pratesh Jayaswal, A. K. Wadhwani and K. B. Mulchandani (2008) Machine Fault Signature Analysis

[20] Elias Houstis (2020) 101 Machine Learning Algorithms for Data Science

[21] L´ eon Bottou, Frank E. Curtis, Jorge Nocedal (2018) Optimization Methods for Large-Scale Machine Learning

[22] Herbert Robbins, Sutton Monro (1951) A Stochastic Approximation Method

[23] Frank & Wolfe (1956) Conditional Gradient

[24] Bertsekas and Tsitsiklis (1989) Parallel coordinate descent and incremental gradient algorithms

[25] Eckstein & Bertsekas (1991) Alternating direction method of multipliers (ADMM)

[26] Hedy Attouch & Juan Peypouquet (2017) The rate of convergence of Nesterov's accelerated forward-backward method is actually faster than 1/k−2

[27] Diederik P. Kingma & Jimmy Lei Ba (2015) ADAM: A method for stochastic optimization

[28] Robert Hecht-Nielsen (1989) Theory of the Backpropagation Neural Network

[29] Pieter-Tjerk De Boer, Dirk P. Kroese, Shie Mannor, Reuven Y. Rubenstein (2004) A Tutorial on the Cross-Entropy Method

[30] Levent Eren & Turker Ince & Serkan Kiranyaz (2017) A Generic Intelligent Bearing Fault Diagnosis System Using Compact Adaptive 1D CNN Classifier

[31] Anirudha Ghosh, Abu Sufian, Farhana Sultana, Amlan Chakrabarti, Debashis De (2020) Fundamental Concepts of Convolutional Neural Network

[32] I. Halil Ozcan, Levent Eren, Turker Ince, Bulent Bilir, Murat Askar (2019) Comparison of time-domain and time-scale data in bearing fault detection

[33] Theodoros Evgeniou and Massimiliano Pontil (2001) Workshop on Support Vector Machines: Theory and applications

[34] B. Samanta, K.R. Al-Balushi, S.A. Al-Araimi (2008) Artificial neural networks and support vector machines with genetic algorithm for bearing fault detection

[35] Levent Eren & Turker Ince & Serkan Kiranyaz (2017) A Generic Intelligent Bearing Fault Diagnosis System Using Compact Adaptive 1D CNN Classifier

[36] P. Konar, P. Chattopadhyay (2011) Bearing fault detection of induction motor using wavelet and Support Vector Machines (SVMs)

[37] Levent Eren (2017) Bearing Fault Detection by One-Dimensional Convolutional Neural Networks

[38] Yuanhong Chang, Jinglong Chen, Cheng Qu, Tongyang Pan (2020) Intelligent fault diagnosis of Wind Turbines via a Deep Learning Network Using Parallel Convolution Layers with Multi-Scale Kernels

[39] Laurens van der Maaten, Geoffrey Hinton (2008) Visualizing Data using t-SNE

Internet resources:

[I] https://dewesoft.com/daq/measure-shock-vibration-with-accelerometers

[II] https://lastminuteengineers.com/adxl335-accelerometer-arduino-tutorial/

[III] https://csegroups.case.edu/bearingdatacenter/home

[IV] https://www.youtube.com/channel/UCYO_jab_esuFRV4b17AJtAw

[V] https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1

[VI] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[VII] https://books.google.com/ngrams

[VIII] https://towardsdatascience.com/the-wavelet-transform-e9cfa85d7b34

[IX] https://www.macnica.co.jp/business/ai_iot/columns/135112/

# Media References

[A] https://www.processingmagazine.com/pumps-motors-drives/bearings-seals/article/15587814/sensors-for-mounted-bearings

[B] https://www.hackerearth.com/blog/developers/3-types-gradient-descent-algorithms-small-large-data-sets/

[C] https://www.hydropower-dams.com/news/reducing-maintenance-with-water-lubricated-turbine-guide-bearings/

# Appendix A – MATLAB and Python Code for Visualization

```
%SCRIPT TO PLOT THE RAW DATA FROM THE CWRU
format compact
clear all
clc

%load all my data files
load('../vis_data/outer_0.021icn_1hp_1772rpm_6oclock.
mat');
xInner = X235_DE_time;

%moving to the right time scale (sample rate 12khz)
fsInner = 12000;
tInner = (0:length(xInner)-1)/fsInner;

%plotting for acceleration vs time
figure
plot(tInner, xInner)
xlabel('Time, (s)')
ylabel('Acceleration (g)')
title('Raw Signal: Normal bearing')
%zoom in the first 0.1sec to see the form of the
singal better
xlim([0 0.5])

%calling matlab commands to move to the frequency
domain through an envelope spectrum diagramm
[pEnvInner, fEnvInner, xEnvInner, tEnvInner] =
envspectrum(xInner, fsInner);
figure
plot(fEnvInner, pEnvInner)
xlim([0 900])
```

Script 1. Visualization of raw acceleration vs time and vs frequency (MATLAB).

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-5,5,100)

y = 1/(1+np.exp(-x))

plt.plot(x,y)
plt.show()
```

Script 2. Visualization of sigmoid activation function (Python).

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-5,5,100)

y = (2/(1+np.exp(-x)))-1

plt.plot(x,y)
plt.show()
```

Script 3. Visualization of tanh activation function (Python).

```
import matplotlib.pyplot as plt
import numpy as np

def relu(X):
  return np.maximum(0,X)

x = np.linspace(-5,5,100)

y = relu(x)

plt.plot(x,y)
plt.show()
```

Script 4. Visualization of ReLU activation function (Python).

```
import matplotlib.pyplot as plt
import numpy as np

def softmax(X):
   expo = np.exp(X)
   expo_sum = np.sum(np.exp(X))
   return expo/expo_sum

x = np.linspace(-5,5,100)

y = softmax(x)

plt.plot(x,y)
plt.show()
```

Script 5. Visualization of SoftMax activation function (Python).

```
import pywt
import matplotlib.pyplot as plt

[phi, psi, x] = pywt.Wavelet('db2').wavefun(level=4)
```

```
plt.plot(x, psi)
```

Script 6. Visualization of Debauche2 wavelet (Python).

```
import pywt
import matplotlib.pyplot as plt

[phi, psi, x] = pywt.Wavelet('db10').wavefun(level=4)

plt.plot(x, psi)
```

Script 7. Visualization of Debauche10 wavelet (Python).

```
lb = -5;
ub = 5;
N = 1000;
[psi,xval] = mexihat(lb,ub,N);
plot(xval,psi)
title('Mexican Hat Wavelet')
```

Script 8. Visualization of Mexican Hat wavelet (MATLAB).

```
lb = -4;
ub = 4;
n = 1000;
[psi,xval] = morlet(lb,ub,n);
plot(xval,psi)
grid on
title('Morlet Wavelet')
```

Script 9. Visualization of Morlet wavelet (MATLAB).

```
import numpy as np
import pywt
import matplotlib.pyplot as plt

wav = pywt.ContinuousWavelet('cmor1.5-1.0')

width = wav.upper_bound - wav.lower_bound

scales = [1, 2, 3, 4, 10, 15]

max_len = int(np.max(scales)*width + 1)
t = np.arange(max_len)
fig, axes = plt.subplots(len(scales), 2, figsize=(12, 6))
for n, scale in enumerate(scales):

   # The following code is adapted from the internals of cwt
   int_psi, x = pywt.integrate_wavelet(wav, precision=10)
```

```
    step = x[1] - x[0]
    j = np.floor(
        np.arange(scale * width + 1) / (scale * step))
    if np.max(j) >= np.size(int_psi):
        j = np.delete(j, np.where((j >= np.size(int_psi)))[0])
    j = j.astype(np.int_)

    # normalize int_psi for easier plotting
    int_psi /= np.abs(int_psi).max()

    # discrete samples of the integrated wavelet
    filt = int_psi[j][::-1]

    nt = len(filt)
    t = np.linspace(-nt//2, nt//2, nt)
    axes[n, 0].plot(t, filt.real)
    axes[n, 0].set_xlim([-max_len//2, max_len//2])
    axes[n, 0].set_ylim([-1, 1])
    axes[n, 0].text(50, 0.35, 'scale = {}'.format(scale))
```

Script 10. Visualization of different scales for the same wavelet (Python).

```
import scipy.io as sio
import numpy as np
from sklearn.manifold import TSNE
import seaborn as sns
palette = sns.color_palette("bright", 4)

# load training data, testing data, validation data and their respective labels
# change file paths and reshape sizes for shallow or deep training
training_dataX =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007_shallo
w/training_dataXs.mat')
training_dataX = np.array(training_dataX['training_dataX']).reshape(212, 200)
training_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007_shallo
w/training_dataYs.mat')
training_dataY = np.array(training_dataY['training_dataY']).reshape(212,)

testing_dataX =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007_shallo
w/testing_dataXs.mat')
testing_dataX = np.array(testing_dataX['testing_dataX']).reshape(92, 200)
testing_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007_shallo
w/testing_dataYs.mat')
testing_dataY1 = np.array(testing_dataY['testing_dataY']).reshape(92,)
```

```
validation_dataX =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.021_shallow/testing_dataXs.mat')
validation_dataX = np.array(validation_dataX['testing_dataX']).reshape(92, 200)
validation_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.021_shallow/testing_dataYs.mat')
validation_dataY = np.array(validation_dataY['testing_dataY']).reshape(92,)

X_embedded = TSNE(n_components=2, perplexity=25,
n_iter=5000).fit_transform(testing_dataX)
X_embedded.shape

sns.scatterplot(X_embedded[:,0], X_embedded[:,1], hue=testing_dataY1, legend='full',
palette=palette)
```

Script 11. T-SNE visualization of the raw training data set before classification.

## Appendix B – MATLAB Code for EFSD Implementation

```
format compact
clear all
clc

load('../../../data/1797_0_0.007/inner.mat');
%load all my data files

xInner = X105_DE_time;
fsInner = 12000;

[pEnvInner, fEnvInner, xEnvInner, tEnvInner] =
envspectrum(xInner, fsInner); %calling matlab comants
to move to the frequency domain through an envelope
spectrum diagramm
figure
plot(fEnvInner, pEnvInner)
%plotting fro amplitude vs frequency
xlim([0 900])
ncomb = 10;
helperPlotCombs(ncomb, 162.077)
%plotting the fundamental fault frequency of the
bearing and its harmonics
xlabel('Frequency (Hz)')
ylabel('Peak Amplitude')
title('Envelope Spectrum: Inner Fault Bearing')
legend('Envelope Spectrum', 'Inner Fault Harmonics')

function helperPlotCombs(ncomb, f)
%defining the helper function to plot the fault
frequencies

ylimit = get(gca, 'YLim');
ylim(ylimit);
ycomb = repmat([ylimit nan], 1, ncomb);
hold(gca, 'on')
for i = 1:length(f)
    xcomb = f(i)*(1:ncomb);
    xcombs = [xcomb; xcomb; nan(1, ncomb)];
    xcombs = xcombs(:)';
    plot(xcombs, ycomb, '--')
end
hold(gca, 'off')
end
```

Script 12. EFSD Implementation for Inner Fault (MATLAB).

```
format compact
clear all
clc

load('../../../data/1797_0_0.021/outer.mat');
%load all my data files

xInner = X234_DE_time;
fsInner = 12000;

[pEnvInner, fEnvInner, xEnvInner, tEnvInner] =
envspectrum(xInner, fsInner); %calling matlab comants
to move to the frequency domain through an envelope
spectrum diagramm
figure
plot(fEnvInner, pEnvInner)
%plotting fro amplitude vs frequency
xlim([0 900])
ncomb = 10;
helperPlotCombs(ncomb, 107.293)
%plotting the fundamental fault frequency of the
bearing and its harmonics
xlabel('Frequency (Hz)')
ylabel('Peak Amplitude')
title('Envelope Spectrum: Outer Fault Bearing')
legend('Envelope Spectrum', 'Outer Fault Harmonics')

function helperPlotCombs(ncomb, f)
%defining the helper function to plot the fault
frequencies

ylimit = get(gca, 'YLim');
ylim(ylimit);
ycomb = repmat([ylimit nan], 1, ncomb);
hold(gca, 'on')
for i = 1:length(f)
    xcomb = f(i)*(1:ncomb);
    xcombs = [xcomb; xcomb; nan(1, ncomb)];
    xcombs = xcombs(:)';
    plot(xcombs, ycomb, '--')
end
hold(gca, 'off')
end
```

Script 13. EFSD Implementation for Outer Fault (MATLAB).

```
format compact
```

```
clear all
clc

load('../../../data/1797_0_0.007/ball.mat');
%load all my data files

xInner = X118_DE_time;
fsInner = 12000;

[pEnvInner, fEnvInner, xEnvInner, tEnvInner] =
envspectrum(xInner, fsInner); %calling matlab comants
to move to the frequency domain through an envelope
spectrum diagramm
figure
plot(fEnvInner, pEnvInner)
%plotting fro amplitude vs frequency
xlim([0 900])
ncomb = 10;
helperPlotCombs(ncomb, 70.5375)
%plotting the fundamental fault frequency of the
bearing and its harmonics
xlabel('Frequency (Hz)')
ylabel('Peak Amplitude')
title('Envelope Spectrum: Ball Fault Bearing')
legend('Envelope Spectrum', 'Ball Fault Harmonics')

function helperPlotCombs(ncomb, f)
%defining the helper function to plot the fault
frequencies

ylimit = get(gca, 'YLim');
ylim(ylimit);
ycomb = repmat([ylimit nan], 1, ncomb);
hold(gca, 'on')
for i = 1:length(f)
    xcomb = f(i)*(1:ncomb);
    xcombs = [xcomb; xcomb; nan(1, ncomb)];
    xcombs = xcombs(:)';
    plot(xcombs, ycomb, '--')
end
hold(gca, 'off')
end
```

Script 14. EFSD Implementation for Outer Fault (MATLAB).

# Appendix C – MATLAB and Python Code for Data Preparation

```
% preproccess for cnn
format compact
clear all
clc
%load all my data files and change the paths and
X***_DE_time lables accordingly
load('..\..\..\data\1797_0_0.007\inner.mat');
load('..\..\..\data\1797_0_0.007\ball.mat');
load('..\..\..\data\1797_0_0.007\outer.mat');
load('..\..\..\data\normal.mat');

inner = X105_DE_time;
ball = X118_DE_time;
outer = X130_DE_time;
normal = X097_DE_time;
%organise every data file in a 605*200 matrix
for i = 1:605
    k = 200*i;
    l = k-199;
    data_i(:,i) = inner(l:k);
    data_b(:,i) = ball(l:k);
    data_o(:,i) = outer(l:k);
    data_n(:,i) = normal(l:k);
end
%decimate every 8th column from these matrices
data_i = transpose(data_i(:,1:6:end));
data_b = transpose(data_b(:,1:6:end));
data_o = transpose(data_o(:,1:6:end));
data_n = transpose(data_n(:,1:6:end));

% keep only the first 53 samples of each class to
constract the training dataset
training_data_i = data_i(1:53,:);
training_data_b = data_b(1:53,:);
training_data_o = data_o(1:53,:);
training_data_n = data_n(1:53,:);

% organise them into one matrix and save them
training_dataX=vertcat(training_data_n,training_data_
i,training_data_o,training_data_b);
a=size(training_dataX,1);
save('..\..\..\data\train_data\0.007_shallow\training
_dataXs.mat','training_dataX')
```

```
% constract the matrix with the labels for the data
for i=1:a
    b=a/4;
    c=2*a/4;
    d=3*a/4;
    training_dataY(1:b,1)=double(0);
    training_dataY(b+1:c,1)=double(1);
    training_dataY(c+1:d,1)=double(2);
    training_dataY(d+1:a,1)=double(3);
end

save('..\..\..\data\train_data\0.007_shallow\training
_dataYs.mat','training_dataY')

% constract the testing data set
testing_data_i = data_i(54:77,:);
testing_data_b = data_b(54:77,:);
testing_data_o = data_o(54:77,:);
testing_data_n = data_n(54:77,:);

% concatenate the matrixes and save them
testing_dataX=vertcat(testing_data_n,testing_data_i,t
esting_data_o,testing_data_b);
a=size(testing_dataX,1);
save('..\..\..\data\train_data\0.007_shallow\testing_
dataXs.mat','testing_dataX')

% constract the label's matrix for the testing data
for i=1:a
    b=a/4;
    c=2*a/4;
    d=3*a/4;
    testing_dataY(1:b,1)=double(0);
    testing_dataY(b+1:c,1)=double(1);
    testing_dataY(c+1:d,1)=double(2);
    testing_dataY(d+1:a,1)=double(3);
end

save('..\..\..\data\train_data\0.007_shallow\testing_
dataYs.mat','testing_dataY')

% constract the testing data set
validation_data_i = data_i(78:101,:);
validation_data_b = data_b(78:101,:);
validation_data_o = data_o(78:101,:);
validation_data_n = data_n(78:101,:);
```

```
% concatenate the matrixes and save them
validation_dataX=vertcat(validation_data_n,validation
_data_i,validation_data_o,validation_data_b);
a=size(validation_dataX,1);
save('..\..\..\data\train_data\0.007_shallow\validati
on_dataXs.mat','validation_dataX')

% constract the label's matrix for the testing data
for i=1:a
    b=a/4;
    c=2*a/4;
    d=3*a/4;
    validation_dataY(1:b,1)=double(0);
    validation_dataY(b+1:c,1)=double(1);
    validation_dataY(c+1:d,1)=double(2);
    validation_dataY(d+1:a,1)=double(3);
end

save('..\..\..\data\train_data\0.007_shallow\validati
on_dataYs.mat','validation_dataY')
```

Script 15. Data preparation for training/validation/testing dataset (MATLAB).

```
import scipy.io as sio
from scipy import stats
import pywt
import math
import numpy as np

def rmsValue(arr, n):
    square = 0
    mean = 0.0
    root = 0.0

    #Calculate square
    for i in range(0,n):
        square += (arr[i]**2)

    #Calculate Mean
    mean = (square / (float)(n))

    #Calculate Root
    root = math.sqrt(mean)

    return root
```

```
def crest_factorValue(x):
    # calculate the crest factor of a signal
    crest_factor = np.max(np.abs(x))/rmsValue(x, len(x))
    return crest_factor

def featureExtraction(data_path, dataset_name):
    # load the .mat data in a pre-arranged matrix
    dataX = sio.loadmat(data_path)
    dataX1 = dataX[dataset_name]

    # define the scales of the transform and the type of wavelet
    scales = np.arange(1, 9)
    wavelet = 'morl'
    shape1 = dataX1.shape[0]
    res = []

    # use wavelet transform on the data and store the resulted coefficients
    for i in range(0,shape1):
        coeffs, freqs = pywt.cwt(dataX1[i], scales, wavelet = wavelet)
        res.append(coeffs)

    # create the features matrix using rms, crest and kurtosis values for each scale of
each sample
    features = []
    for i in range(0, shape1):
        for j in range(0, 8):
            rms = rmsValue(res[i][j], len(res[i][j]))
            crest = crest_factorValue(res[i][j])
            kurt = stats.kurtosis(res[i][j])
            features.append(rms)
            features.append(crest)
            features.append(kurt)

    # format the features matrix in a 24*shape1 matrix, ready for input in the network
    struct_features = np.array(features).reshape((shape1, 24))
    return struct_features
```

Script 16. Feature extraction using Continuous Wavelet Transform (Python).

# Appendix D – Python Code for Intelligent Methods

```
''' Create multi-class SVM with sklearn and python '''
# import all necessary modules
import scipy.io as sio
import numpy
import time
from sklearn import svm
from sklearn import metrics
from sklearn.metrics import confusion_matrix
from keras.utils import np_utils
import seaborn as sn
import matplotlib.pyplot as plt
import sys
import numpy as np
from sklearn.manifold import TSNE
palette = sn.color_palette("bright", 4)

sys.path.append('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/CWRU/scripts/dat
a_preprocess')
from wavelet_transform import featureExtraction

# initialize a time counter and a random seed to ensure reproducability
start_time = time.time()
np.random.seed(7)

# load training data, testing data and their respective labels
training_dataX =
featureExtraction('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/
0.007_shallow/training_dataXs.mat', 'training_dataX')
training_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/training_dataYs.mat')
training_dataY = np.array(training_dataY['training_dataY']).reshape(212,)

testing_dataX =
featureExtraction('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/
0.007_shallow/testing_dataXs.mat', 'testing_dataX')
testing_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/testing_dataYs.mat')
testing_dataY1 = np.array(testing_dataY['testing_dataY']).reshape(96,)
```

```
validation_dataX =
featureExtraction('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/
0.007_shallow/validation_dataXs.mat', 'validation_dataX')
validation_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/validation_dataYs.mat')
validation_dataY = np.array(validation_dataY['validation_dataY']).reshape(96,)

# import the SVM from sklearn and define RBF as the kernel function
classification = svm.SVC(kernel='rbf')

# train the SVM on the training dataset
classification.fit(training_dataX, training_dataY)

testing_dataY2 = np_utils.to_categorical(testing_dataY1)
validation_dataY2 = np_utils.to_categorical(validation_dataY)

# make prediction based on the trained model
Y_predict = classification.predict(testing_dataX)
Y_predict = Y_predict.reshape(Y_predict.shape[0],)
print("Accuracy Validation:", metrics.accuracy_score(testing_dataY1, Y_predict))
print(metrics.classification_report(testing_dataY1, Y_predict))
# calculate and print the time needed for training
end_time = time.time()
execution_time = end_time - start_time
print('SVM Training time(in sec): %.2f' % execution_time)

# make predictions for the testing data and constract the confusion matrix
conf_matrix = confusion_matrix(testing_dataY1, Y_predict)
print('Cofusion Matrix for the test data:')
print(conf_matrix)

# plot the heatpam of the confusion matrix
sn.heatmap(conf_matrix, annot=True, fmt='g', annot_kws={"size": 16},
cmap="YlGnBu")
plt.show()

# make prediction based on the trained model
Y_predict2 = classification.predict(validation_dataX)
Y_predict2 = Y_predict2.reshape(Y_predict2.shape[0],)
print("Accuracy Test:", metrics.accuracy_score(validation_dataY, Y_predict2))
print(metrics.classification_report(validation_dataY, Y_predict2))

# make predictions for the validation data and constract the confusion matrix
```

```
conf_matrix2 = confusion_matrix(validation_dataY, Y_predict2)
print('Cofusion Matrix for the validation data:')
print(conf_matrix2)

# plot the heatpam of the confusion matrix
sn.heatmap(conf_matrix2, annot=True, annot_kws={"size": 16}, cmap="YlGnBu")
plt.show()

# time each prediction
time_test_sample = validation_dataX[0]
time_test_sample = time_test_sample.reshape(1,24)

prediction_time_s = time.time()
Y_predict3 = classification.predict(time_test_sample)
prediction_time_e = time.time()
print('Time per sample', prediction_time_e - prediction_time_s)

# use t-SNE to visualize output
vis_preds = classification.predict(testing_dataX)
vis_preds = np.array(vis_preds).reshape(96, 1)

X_embedded = TSNE(n_components=2, perplexity=50,
n_iter=1000).fit_transform(vis_preds)
X_embedded.shape

sn.scatterplot(X_embedded[:,0], X_embedded[:,1], hue=testing_dataY1,
legend='full', palette=palette)
```

Script 17. SVM Implementation (Python).

```
''' Create multi-class MLP with Keras and python '''
# import all necessary modules
from keras.models import Sequential, Model
from keras.layers import Dense
import seaborn as sn
from sklearn.metrics import confusion_matrix
from sklearn import metrics
import matplotlib.pyplot as plt
import scipy.io as sio
import numpy as np
import sys
import time
from sklearn.preprocessing import LabelEncoder
from keras.utils import np_utils
from sklearn.manifold import TSNE
```

```
palette = sn.color_palette("bright", 4)

sys.path.append('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/CWRU/scripts/dat
a_preprocess')
from wavelet_transform import featureExtraction

# initialize a time counter and a random seed to ensure reproducability
start_time = time.time()
np.random.seed(7)

# load training data, testing data and their respective labels
# change file paths and reshape sizes for shallow or deep training
training_dataX =
featureExtraction('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/
0.007_shallow/training_dataXs.mat', 'training_dataX')
training_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/training_dataYs.mat')
training_dataY1 = np.array(training_dataY['training_dataY']).reshape(212,)

testing_dataX =
featureExtraction('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/
0.007_shallow/testing_dataXs.mat', 'testing_dataX')
testing_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/testing_dataYs.mat')
testing_dataY = np.array(testing_dataY['testing_dataY']).reshape(96,)

validation_dataX =
featureExtraction('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/
0.007_shallow/validation_dataXs.mat', 'validation_dataX')
validation_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/validation_dataYs.mat')
validation_dataY1 = np.array(validation_dataY['validation_dataY']).reshape(96,)

# encode class values as integers
encoder = LabelEncoder()
encoder.fit(training_dataY1)
encoded_Y = encoder.transform(training_dataY1)
# convert integers to dummy variables (i.e. one hot encoded)
training_dataY = np_utils.to_categorical(encoded_Y)

encoder2 = LabelEncoder()
```

```
encoder2.fit(validation_dataY1)
encoded_Y = encoder2.transform(validation_dataY1)
# convert integers to dummy variables (i.e. one hot encoded)
validation_dataY = np_utils.to_categorical(encoded_Y)

# define the multiclass MLP
def baseline_model():
    model = Sequential()
    model.add(Dense(8, input_dim=24, activation='tanh'))
    model.add(Dense(9, activation='tanh'))
    model.add(Dense(8, activation='tanh'))
    model.add(Dense(4, activation='softmax'))
    return model

# Compile model
model = baseline_model()
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
# Fit the model
his = model.fit(training_dataX, training_dataY, validation_data=(validation_dataX,
validation_dataY), epochs=200, batch_size=100)

# evaluate the model
scores = model.evaluate(validation_dataX, validation_dataY)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

# plot the training and validation loss
plt.plot(his.history['loss'], label="Train")
plt.plot(his.history['val_loss'], label="Test")
plt.legend(loc="upper right")
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()

# calculate and print the time needed for training
end_time = time.time()
execution_time = end_time - start_time
print('MLP Training time(in sec): %.2f' % execution_time)

# make predictions for the validation data and constract the confusion matrix
predictions = model.predict_classes(validation_dataX, batch_size=100, verbose=0)
conf_matrix = confusion_matrix(validation_dataY1, predictions)
print('Cofusion Matrix for the validation data:')
print(conf_matrix)

# plot the heatpam of the confusion matrix
sn.heatmap(conf_matrix, annot=True, fmt='g', annot_kws={"size": 16},
cmap="YlGnBu")
plt.show()

# make predictions for the testing data and constract the confusion matrix
predictions2 = model.predict_classes(testing_dataX, batch_size=100, verbose=0)
conf_matrix2 = confusion_matrix(testing_dataY, predictions2)
print('Cofusion Matrix for the validation data:')
print(conf_matrix2)

# print full metrics for the test data
print("Accuracy Test:",metrics.accuracy_score(testing_dataY, predictions2))
print(metrics.classification_report(testing_dataY, predictions2))

# plot the heatpam of the confusion matrix for the test data
sn.heatmap(conf_matrix2, annot=True, fmt='g', annot_kws={"size": 16},
cmap="YlGnBu")
plt.show()

# time each prediction
time_test_sample = validation_dataX[0]
time_test_sample = time_test_sample.reshape(1,24)

prediction_time_s = time.time()
Y_predict3 = model.predict_classes(time_test_sample)
prediction_time_e = time.time()
print('Time per sample', prediction_time_e - prediction_time_s)

# use t-SNE to visualize output
model2 = Model(model.input, model.layers[3].output)
vis_preds=model2.predict(testing_dataX)

X_embedded = TSNE(n_components=2, perplexity=30,
n_iter=5000).fit_transform(vis_preds)
X_embedded.shape

sn.scatterplot(X_embedded[:,0], X_embedded[:,1], hue=testing_dataY, legend='full',
palette=palette)
```

Script 18. MLP Implementation (Python).

```
''' Create CNN with Keras and python '''
```

```
# import all necessary modules
import numpy
import keras
import time
import scipy.io as sio
import seaborn as sn
import numpy as np
from sklearn import metrics
from sklearn.metrics import confusion_matrix
from keras.models import Sequential, Model
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.utils import np_utils
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE

palette = sn.color_palette("bright", 4)

# define a callback class to end training when accuracy of 98% is reached
class myCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        e=0.98
        if(logs.get('accuracy') > e):
            print("\nReached %2.2f%% accuracy in last epoch, so stopping training!!"
%(e*100))
            self.model.stop_training = True
Callback = myCallback()

# initialize a time counter and a random seed to ensure reproducability
start_time = time.time()
numpy.random.seed(7)

# load training data, testing data, validation data and their respective labels
# change file paths and reshape sizes for shallow or deep training
training_dataX =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/training_dataXs.mat')
training_dataX = np.array(training_dataX['training_dataX']).reshape(212, 200, 1)
training_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/training_dataYs.mat')
training_dataY1 = np.array(training_dataY['training_dataY']).reshape(212,)
```

```
testing_dataX =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/testing_dataXs.mat')
testing_dataX = np.array(testing_dataX['testing_dataX']).reshape(96, 200, 1)
testing_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/testing_dataYs.mat')
testing_dataY = np.array(testing_dataY['testing_dataY']).reshape(96,)

validation_dataX =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/validation_dataXs.mat')
validation_dataX = np.array(validation_dataX['validation_dataX']).reshape(96, 200,
1)
validation_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/validation_dataYs.mat')
validation_dataY1 = np.array(validation_dataY['validation_dataY']).reshape(96,)

# convert integers to dummy variables (i.e. one hot encoded)
training_dataY = np_utils.to_categorical(training_dataY1)
validation_dataY = np_utils.to_categorical(validation_dataY1)

# take the input dimensions from the data
timesteps = training_dataX.shape[1]
features = training_dataX.shape[2]

# define the CNN
def baseline_model():
   model = Sequential()

model.add(Conv1D(60,9,input_shape=(timesteps,features),activation='tanh',padding
='same'))
   model.add(MaxPooling1D(4))
   model.add(Conv1D(40,9,activation='tanh',padding='same'))
   model.add(MaxPooling1D(4))
   model.add(Conv1D(40,9,activation='tanh',padding='same'))
   model.add(Flatten())
   model.add(Dense(20,activation='tanh'))
   model.add(Dense(4,activation='softmax'))
   model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
   return model
```

```
# compile model and start training
model = baseline_model()
history = model.fit(training_dataX, training_dataY,
validation_data=(validation_dataX, validation_dataY), epochs=15, batch_size=100,
callbacks=[Callback])

scores = model.evaluate(validation_dataX, validation_dataY)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

# plot the training and testing loss
plt.plot(history.history['loss'], label="Train")
plt.plot(history.history['val_loss'], label="Test")
plt.legend(loc="upper right")
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()

# calculate and print the time needed for training
end_time = time.time()
execution_time = end_time - start_time
print('CNN Training time(in sec): %.2f' % execution_time)

# make predictions for the validation data and constract the confusion matrix
predictions = model.predict_classes(validation_dataX, batch_size=100, verbose=0)
print("Accuracy Val:",metrics.accuracy_score(validation_dataY1, predictions))
print(metrics.classification_report(validation_dataY1, predictions))
conf_matrix = confusion_matrix(validation_dataY1, predictions)
print('Cofusion Matrix for the test data:')
print(conf_matrix)

# make predictions for the testing data and constract the confusion matrix
predictions2 = model.predict_classes(testing_dataX, batch_size=100, verbose=0)
conf_matrix2 = confusion_matrix(testing_dataY, predictions2)
print('Cofusion Matrix for the validation data:')
print(conf_matrix2)

# print metrics of testing data
print("Accuracy Test:",metrics.accuracy_score(testing_dataY, predictions2))
print(metrics.classification_report(testing_dataY, predictions2))

# plot the heatpam of the confusion matrix
```

```
sn.heatmap(conf_matrix, annot=True, fmt='g', annot_kws={"size": 16},
cmap="YlGnBu")
plt.show()

# plot the heatpam of the confusion matrix
sn.heatmap(conf_matrix2, annot=True, annot_kws={"size": 16}, cmap="YlGnBu")
plt.show()

# time each prediction
time_test_sample = validation_dataX[0]
time_test_sample = time_test_sample.reshape(1,200,1)

prediction_time_s = time.time()
Y_predict3 = model.predict_classes(time_test_sample)
prediction_time_e = time.time()
print('Time per sample', prediction_time_e - prediction_time_s)

# use t-SNE to visualize output
testing_dataX2 = np.array(validation_dataX).reshape(96, 200)
model2 = Model(model.input,model.layers[7].output)
vis_preds=model2.predict(testing_dataX2)

X_embedded = TSNE(n_components=2, perplexity=30,
n_iter=5000).fit_transform(vis_preds)
X_embedded.shape

sn.scatterplot(X_embedded[:,0], X_embedded[:,1], hue=validation_dataY1,
legend='full', palette=palette)
```

Script 19. CNN Implementation (Python).

```
''' Create C-CNN with Keras and python '''
# import all necessary modules
import numpy
import time
import scipy.io as sio
from sklearn import metrics
from sklearn.metrics import confusion_matrix
from keras import Model
from keras import Input
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Concatenate
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.utils import np_utils
import matplotlib.pyplot as plt
import seaborn as sn
import numpy as np
from sklearn.manifold import TSNE

palette = sn.color_palette("bright", 4)

# initialize a time counter and a random seed to ensure reproducability
start_time = time.time()
numpy.random.seed(7)

# load training data, testing data, validation data and their respective labels
# change file paths and reshape sizes for shallow or deep training
training_dataX =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/training_dataXs.mat')
training_dataX = np.array(training_dataX['training_dataX']).reshape(212, 200, 1)
training_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/training_dataYs.mat')
training_dataY1 = np.array(training_dataY['training_dataY']).reshape(212,)

testing_dataX =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/testing_dataXs.mat')
testing_dataX = np.array(testing_dataX['testing_dataX']).reshape(96, 200, 1)
testing_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/testing_dataYs.mat')
testing_dataY = np.array(testing_dataY['testing_dataY']).reshape(96,)

validation_dataX =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/validation_dataXs.mat')
validation_dataX = np.array(validation_dataX['validation_dataX']).reshape(96, 200,
1)
validation_dataY =
sio.loadmat('C:/Users/ilias/Desktop/MHX.MHX/διπλωματικη/data/train_data/0.007
_shallow/validation_dataYs.mat')
validation_dataY1 = np.array(validation_dataY['validation_dataY']).reshape(96,)
```

```
# convert integers to dummy variables (i.e. one hot encoded)
training_dataY = np_utils.to_categorical(training_dataY1)
validation_dataY = np_utils.to_categorical(validation_dataY1)

# take the input dimensions from the data
timesteps = training_dataX.shape[1]
features = training_dataX.shape[2]

# define the number of filters, the different filter sizes and the number of different
branches
n_filters = 64
filter_size = [5, 25, 50, 100, 125]
input_shape = (timesteps, features)
pool_size = 10
n_paraller_branches = 5

# define the different parallel branches
inp = Input(shape=input_shape)
convolutions = []
for k in range(len(filter_size)):
    convolution1 = Conv1D(n_filters, filter_size[k], padding='same', activation='relu',
input_shape=input_shape)(inp)
    pool1 = MaxPooling1D(pool_size=pool_size)(convolution1)
    convolution2 = Conv1D(n_filters, filter_size[k], padding='same',
activation='relu')(pool1)
    pool2 = MaxPooling1D(pool_size=pool_size)(convolution2)
    convolutions.append(pool2)

# feed the output in the concatenation layer
out = Concatenate()(convolutions)

# create the parallel model
conv_model = Model(inp, out)

# define the C-CNN
def baseline_model():
    model = Sequential()
    model.add(conv_model)
    model.add(Flatten())
    model.add(Dense(4, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
    return model
```

```
# start the training of the network
model = baseline_model()
history = model.fit(training_dataX, training_dataY,
validation_data=(validation_dataX, validation_dataY), epochs=20, batch_size=150)

scores = model.evaluate(validation_dataX, validation_dataY)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

# plot the loss
plt.plot(history.history['loss'], label="Train")
plt.plot(history.history['val_loss'], label="Test")
plt.legend(loc="upper right")
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()

# calculate and print the time needed for training
end_time = time.time()
execution_time = end_time - start_time
print('C-CNN Training time(in sec): %.2f' % execution_time)

# make predictions for the validation data and constract the confusion matrix
predictions = model.predict_classes(validation_dataX, batch_size=150, verbose=0)
print(metrics.classification_report(validation_dataY1, predictions))
conf_matrix = confusion_matrix(validation_dataY1, predictions)
print('Cofusion Matrix for the test data:')
print(conf_matrix)

# make predictions for the testing data and constract the confusion matrix
predictions2 = model.predict_classes(testing_dataX, batch_size=150, verbose=0)
conf_matrix2 = confusion_matrix(testing_dataY, predictions2)
print('Cofusion Matrix for the validation data:')
print(conf_matrix2)

# print metrics of validation data
print("Accuracy:",metrics.accuracy_score(testing_dataY, predictions2))
print(metrics.classification_report(testing_dataY, predictions2))

# plot the heatpam of the confusion matrix
sn.heatmap(conf_matrix, annot=True, fmt='g', annot_kws={"size": 16},
cmap="YlGnBu")
plt.show()
```

```
# plot the heatpam of the confusion matrix
sn.heatmap(conf_matrix2, annot=True, fmt='g', annot_kws={"size": 16},
cmap="YlGnBu")
plt.show()

# time each prediction
time_test_sample = validation_dataX[0]
time_test_sample = time_test_sample.reshape(1,200,1)

prediction_time_s = time.time()
Y_predict3 = model.predict_classes(time_test_sample)
prediction_time_e = time.time()
print('Time per sample', prediction_time_e - prediction_time_s)

# use t-SNE to visualize output
testing_dataX2 = np.array(testing_dataX).reshape(96, 200)
model2 = Model(model.input, model.layers[2].output)
vis_preds=model2.predict(testing_dataX2)

X_embedded = TSNE(n_components=2, perplexity=30,
n_iter=1000).fit_transform(vis_preds)
X_embedded.shape

sn.scatterplot(X_embedded[:,0], X_embedded[:,1], hue=testing_dataY, legend='full',
palette=palette)
```

Script 20. C-CNN Implementation (Python).