UNIVERSITY OF DEBRECEN
FACULTY OF ENGINEERING
DEPARTMENT OF
MECHANICAL ENGINEERING

PREDICTIVE MAINTENANCE ON MANUFACTURING LINES USING EDGE-BASED MACHINE LEARNING OPERATIONS

THESIS

ADEEL KAMAL
Production Engineering Specialization

Debrecen
2026

Table of Contents

Introduction

In any manufacturing system certain mechanical components are not interchangeable therefore in order to improve the reliability, product quality, operational safety, and manufacturing cost these components which are of the rolling nature must be maintained at a regular time cycle which are usually set by the original equipment manufacturer or the maintenance team in charge of the assembly line. Among these Rotating Machine's the Bearing is widely used and most faults that occur are linked to bearing defects, which in turn cause unplanned downtime, High Maintenance cost, and as a result productivity is negatively affected. Traditional Maintenance strategies like preventive (and reactive in today's world of instant demand lose their appeal as they wait for a failure to occur or depend on a fixed timing in order to prevent it, this also is a way in which the machine health is not optimally diagnosed.

Predictive maintenance is a technique that collaborates between hardware and software to allow the manufacturer to foresee any upcoming issues that are about to occur. This is done by statistical methods (Machine learning) and the more efficient "deep Learning" way, In which Historical data is collected through sensors for bearing accelerometer can be used to collect Raw Vibrational data ,conduct preprocessing and Feature Engineering on it After Which depending on the nature of data and the result required , the cleaned data is passed through either a ML Algorithm or a Neural networks to predict the faults the could occur. Vibration Analysis that is discussed in further detail in this thesis is one of the most efficient ways to this. The analysis in done on the CWRU Dataset which is publicly available by the Case Western Reserve University (CWRU). this is an established Benchmark for research on bearing diagnostics.

As the industry evolves access to IOT device, Machine learning, and edge computing is much more feasible. These predictive maintenance systems are able to operate in real-time and can be directly deployed on the manufacturing lines. Directly deploying these models on edge devices minimizes the connection requirements, it benefits also bears the advantage that in the case of network disruption your process is not affected. Combining edge inference with cloud-based MLOps (Machine Learning Operations) enables scalable model management, monitoring, and continuous improvement

The aim of this thesis is to develop a complete predictive maintenance workflow using vibration signals. The work includes data preprocessing, feature extraction using envelope analysis, machine learning model training, and the design of an edge–cloud MLOps pipeline for real-time deployment and monitoring.

1. Literature review

Predictive maintenance (PdM) has become a central topic in Industry 4.0, enabling real-time monitoring of machine health through sensor data, data analytics, and machine learning [3]. Unlike traditional maintenance approaches, predictive maintenance focuses on forecasting equipment failures before they occur, reducing both downtime and operational cost [27]. Several studies highlight that vibration-based PdM significantly improves reliability in rotating machinery compared to temperature, acoustic, or electrical monitoring [4].

1.1 Industrial Revolution

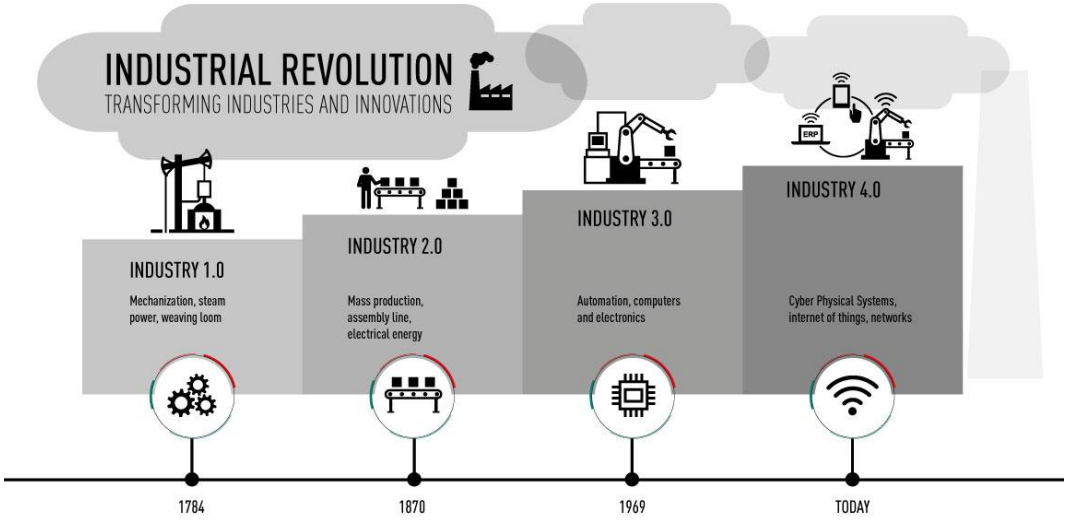

Figure 1 Industrial Revolution Timeline [18].

1.1.1 First Industrial revolution

The invention of the steam power and the mechanization of production led to the start of the first industrial revolution which began in the 18th century [2]. Although historians debate the exact date in which the industrial revolution started most agree with the conclusion that it was in the 18th century in Britain [20]. The first industrial revolution increased productivity 10 folds, what was once produced on spinning wheels the mechanized version could produce 8 times more [12]. The breakthrough that was the steam engine was the greatest achievement of the time and the cause of human propulsion into the new era [1]. Instead of weaving looms that were powered by humans, shifts to steam power was adopted.it reshaped how we travel and conducted logistics by the use of steam powered locomotives [1].

1.1.2 Second Industrial Revolution

By the second industrial revolution (which is dated between 1870 and 1914), the population shift of the population cities had gone up from 6% (in the 18th century) to 40 % [20].the best example of what the 2nd industrial revolution brought about is henry for the founder of ford motors adopting the assembly line, Where the automotives were being assembled partially on assembly lines rather than one being assembled on one entire station[2]. In addition to the rise in urbanization, innovations in the field of electricity, radio and communications shaped and retransformed the way people lived [1].

1.1.3 Third Industrial revolution

The third Industrial Revolution brought about the change of how the overall processes were powered [21]. This revolution introduced computer Technology as an interface which rapidly increased the speed of production as a trade-off it also reduced the importance of human power [1]. It revolved around the invention of PLC programable logic controller that helped in automating processes and the development of digital electronics [1]. These changes had a crucial effect on the world that changed the socio-economic and socio-cultural impression, and it still continues to do so, due to the increase in optimization of the reliability of manufacturing processes increased, that led to the enhancement of industrialization [2].

1.1.4 Fourth Industrial Revolution

The Fourth Industrial Revolution is defined by the emergence of cyber-physical systems, which are emblematic of this wave's entirely new capabilities for individuals and machines [22]. These systems focus on the integration and advancement of knowledge and information systems alongside an exponential increase in computing, transmission, and storage capacity [1]. These systems promote the development of robust, interlinked, and novel technological processes [2].

1.2 Bearing Failure Mechanisms

Machines consist of various parts that need to be maintained at certain intervals to ensure the optimization of the product production [23]. Among these various mechanical parts, the most prone to cause defects are the rolling elements in the system also known as bearing [7]. Bearings go through various loads while operating which can cause defects like cracks or faults in different places [4]. These faults are mainly categorized in the outer race, inner race, cage and ball. These faults arising have huge effects on the overall performance and efficiency of both the factory workers and the production as well due to the compromised stability that faults in these parts because it also effects the life of the machine [8].

Figure 1.2 illustrates the different fault locations that occur in bearings, as previously mentioned the outer race, inner race, ball and cage [3].

Types of faults in circular bearings [9]. As listed below:

- The outer race: usually mounted on the motor cap effects the alignment.
- The inner race: it holds the motor shaft where there may be shaft deflection.
- Ball and cage: these are for restraining the relative distance of the bearing.

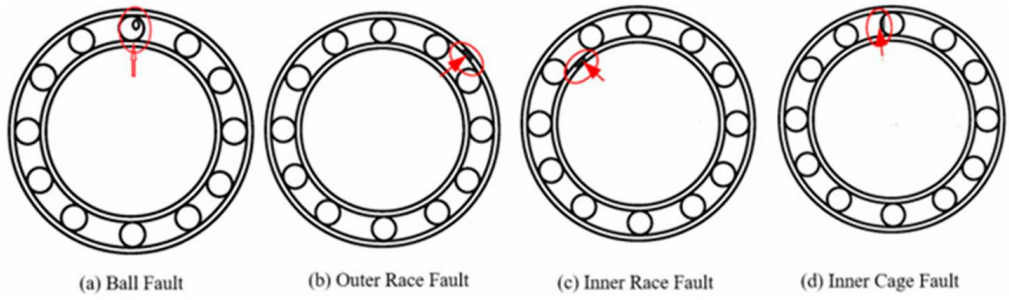

Figure 2 Bearing faults [9].

1.2.1 Vibration Analysis

Bearing defects are the most common defects that occur in machines and are classified under the Rolling element characterization; therefore, a relevant approach is needed to address these faults that occur [27]. The most important and most studied technique used is the use of vibration analysis [5]. This approach is particularly useful as it can detect a fault about to occur in the early stages. Fault detection in the early stages can save up precious time by reducing the planned and unplanned downtimes as well as reduce cost and labour hours per maintenance schedule. Vibration Analysis is proven to work in both rotating and non-rotating parts [5].

Vibration analysis has picked up further popularity with researcher and professional in regards to predicting when a failure might occur [6]. Multiple industries are adapting to predictive maintenance a couple of examples of which industries are using it are Power generation, cement plants, and civil construction industry [11]. Predictive maintenance has borne fruitful results in these and any other industries [19]. Which in term has resulted in more optimal production and safer working and operating conditions [10].

1.2.2 Vibration Analysis Basics

The implementation of Early fault detection is made efficient by combining it with predictive maintenance, it does this by the use of powerful tools like vibration analysis [24]. Any machine under loads emits certain vibrations that can be diagnosed and analysed to detect anomalies which if left unchecked can cause a decrease in efficiency, higher rate of both planned and unplanned downtime,

machine life degradation, and might also be prone to accidents, these vibrations vary and are of different types [10].

When a force is applied at a certain point also known as a reference point the movement that is caused by this force on point is called a vibration. These vibrations occur in forms often classified as random or periodic as well as harmonic or non-harmonic [24]. When a machine is producing its back-and-forth movement during operation it is creating vibrations, and these vibrations carry an amplitude that differs in different types of machines as it Is relative to the load [8]. There is a certain normal amplitude that the machine would produce under normal operating conditions, if the amplitude starts to increase from this normal level, the vibrations that are occurring start to produce wear, tear and fatigue to the machine compromising its overall reliability [4]. It is also important to note that vibrations of different component carry a specific type of vibration, these patterns in the vibrations can be noted for each mechanical component [8].

1.2.3 2.3.2 Vibration Profile

Motion can be categorized into harmonic motion and nonharmonic motion. Harmonic motions are those that recur with each full cycle [25]. Conversely, nonharmonic motions are characterized by the combination of movements from multiple sources that have varying frequencies [10]. In this context, let X1 and X2 represent two vibration displacements with distinct frequencies:

$$X_1 = a\sin(\omega_1 t) \quad (1)$$

$$X_2 = bsin(\omega_2 t) \quad (2)$$

In the equation (1) and (2) the variables „a" and" b" represent the maximum amplitude of the frequency. ω1 and ω2 the in circular frequency and the time is represented by „t" (3).

$$[f(x) = \max(0, x)] \quad (3)$$

This equation (4) known as the Fourier Series equation says that any frequency can be shown as a series of sine functions with circular frequencies.

1. ω1, ω2: are the harmonics of the initial frequency.
2. A1, A2: are the amplitudes of the different frequencies
3. Φ: are there corresponding phase angle.

$$f(t) = A_0 + A_1 \sin(\omega t + \phi_1) + A_2 \sin(2\omega t + \phi_2) + A_3 \sin(3\omega t + \phi_3) + \cdots \quad (4)$$

Vibration profiles have ways to be represented among these ways are the time-domain Figure 1.3 and the frequency-domain Figure 1.4. In the time domain the frequency is shown in time -Amplitude [5].

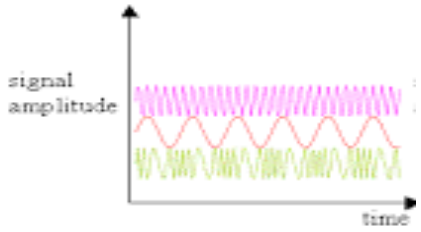

Figure 3 Time Domain [16].

In the Frequency domain figure below, it is shown in the frequency and amplitude.

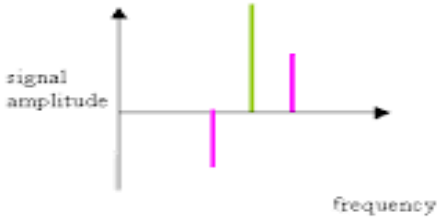

Figure 4 Frequency Domain [16].

1.2.4 The CWRU Test-set up

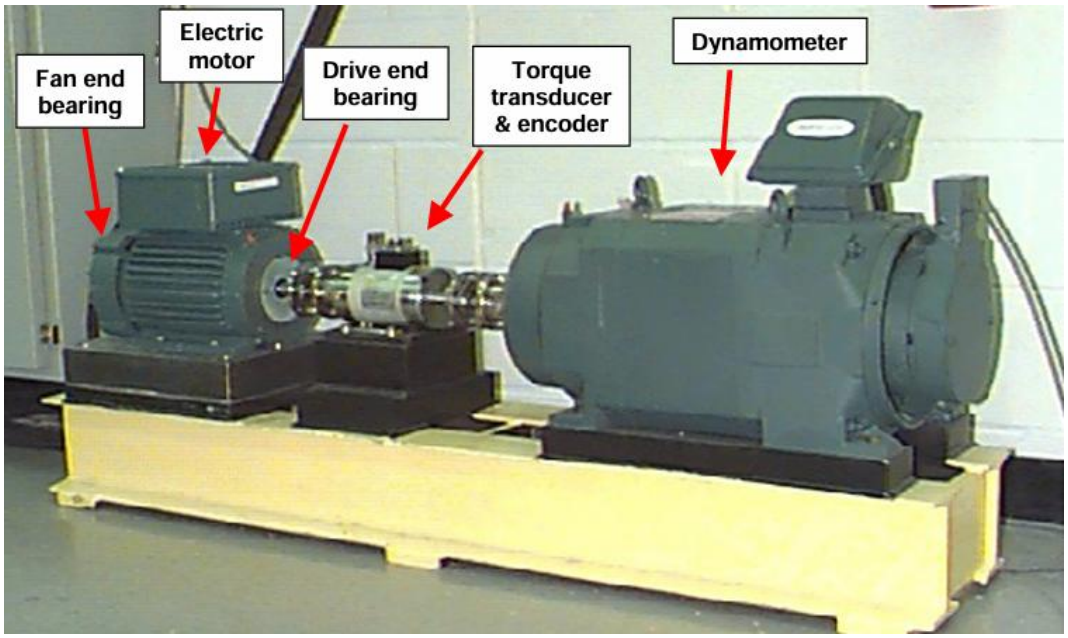

Figure 5 Machine set-up for Bearing data [5].

The case Western Reserve University created a dataset using the above setup (figure 1.5), using Electro discharging machines bearings were installed in two the locations, these mentioned locations were at the drive end and the fan end of the test setup [26]. A 2-horsepower electric motor was used to drive the shaft. On top of said shat a torque transducer and encoder was attached [5]. As for the torque being applied to the shaft, it was done by the dynamometer that is on the left of the figure 1.5 [5].

The faults that were on implanted on the bearings were at the location of the ball, the bearing inner race and the bearing outer race. The loads at which these bearing were used at are ranged from the horsepower of 0 to 3 Hp [6]. The rolling elements are the being put on the machine separately and then being run on the configuration being used in fig.4 [26]. through this technique the reading that are being generated during the running operation of the experiment are being recorded. As for how these reading are being recorded, it is done using sensors, the particular type of sensor that is used here in the configuration in figure 1.4 is the accelerometer [5].

1.2.5 The CWRU Data set configuration

The website of the University that has provided the data gives four files to work with [13]. One file is the Normal Baseline data that lists the normal reading, in these readings the data which is gathered is being gathered when the configuration it operating under optimal environment [6]. What that means is the rolling element

that is used does not have any faults [15]. Also to further add it is also undergoing through the stable load and the speed as well is stable table 1.1 below illustrates the contents of these files [26].

Table 1.1 motor loads in RPM [5].

Load	Motor Speed (Rpm)
0	1796
1	1772
2	1750
3	1730

The second file is the 12k drive end data, in which the fault diameters are introduced [4]. At each fault four (0,1,2,3) loads are added at three different position these positions are the canter ,90 degree, and in the opposing position [4]. All ate at the 3 different positions except the 0.014 and the 0.028 diameter. As shown in table 1.2 below.

Table 1.2 12k drive end data [14].

Fault Diameter	Motor Load (HP)	Approx. Motor Speed (rpm)	Inner Race	Ball	Outer Race Position Relative to Load Zone (Load Zone Centered at 6:00)		
					Centered @6:00	Orthogonal @3:00	Opposite @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	*	*	*

The third file is the 48k Drive End Fault Data in which three fault diameters are used (0.007'',0.014'' and 0.021''). The loads for each fault are at the o to 3 range meaning it is acting at 4 varying loads. As for the load that are being used on the outer race of the bearing it is being done at the 3 different locations of centers, 90 degree and the opposing. It is noteworthy to mention the inner race of the bearing do not have different load zones. It is also important to mention that at the 2nd diameter defect on the outer race the load zone is only at the @6:00 position [14]. Table 1.3 below

Table 1.3 48k Drive End Fault Data [14].

Fault Diameter	Motor Load (HP)	Approx. Motor Speed (rpm)	Inner Race	Ball	Outer Race Position Relative to Load Zone (Load Zone Centered at 6:00)		
					Centered @6:00	Orthogonal @3:00	Opposite @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

The fourth and final configuration is 12k Fan End Bearing Fault Data and its similarly being applied to the 48k Drive End Fault Bearing shown in table 1.5 below.

Table 1.4 12k Fan End Bearing Fault Data [14].

Fault Diameter	Motor Load (HP)	Approx. Motor Speed (rpm)	Inner Race	Ball	Outer Race Position Relative to Load Zone (Load Zone Centered at 6:00)		
					Centered @6:00	Orthogonal @3:00	Opposite @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	*

1.2.6 Signal Processing Methods for Bearing Fault Detection

Raw vibration signals contain noise from mechanical systems, electrical interference, environment, and machine structure [5]. Traditional signal processing techniques aim to enhance defect signatures by suppressing noise [27]. Bandpass filtering isolates high-frequency resonance bands where defect impacts are amplified. The Hilbert transform extracts the envelope of the signal, which represents the amplitude variations caused by defect impacts [6]. The envelope spectrum, computed using the Fast Fourier Transform (FFT), reveals clear peaks at theoretical fault frequencies (BPFO, BPFI, BSF, FTF) [27].

1.2.7 Machine Learning and Deep Learning for Fault Classification

Artificial intelligence is term widely used and it elements consist of more and more precise elements for instant the term AI could be considered a large circle and within this circle consist a more concise circle called Machine learning, which uses algorithms and simple networks for prediction [28]. Now imagine an even more concise circle with in the machine learning world and this is the part where deep

learning works and this technique uses more complex systems to find patterns in complex patterns [29].

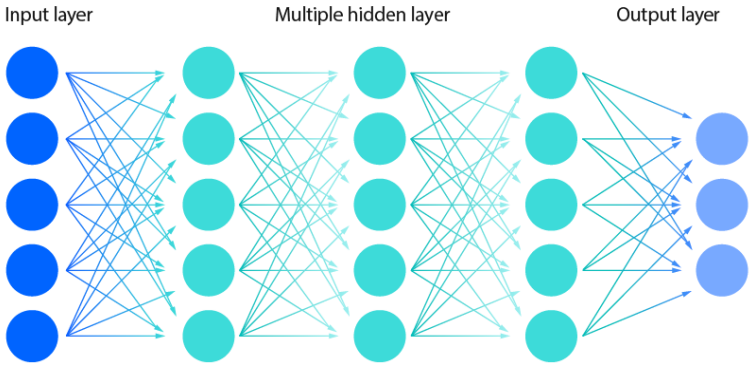

Figure 6 Deep Neural Network Structure [17].

1.2.8 Edge Computing and MLOps in Predictive Maintenance

Edge computing brings computation closer to the source of data, reducing latency and bandwidth usage [30]. In predictive maintenance systems, edge devices perform data preprocessing and model inference in real-time, while cloud platforms handle training, version control, and monitoring [9]. MLOps ensures reliable deployment, scalability, and continuous improvement of ML models [11]. Docker containers, Kubernetes, and lightweight runtime frameworks such as TensorFlow Lite or ONNX Runtime enable practical industrial implementations [10].

2. Collect and preprocess vibration data from CWRU

2.1 Overview

This section explains the methodology that was used to collect the data from the CRWU dataset which is available on the case western reserve university website. The Data used in this paper was instead downloaded from the Kaggle website due to technical issues with the original dataset source, but the Kaggle data structure is completely authentic and is widely adopted.2 data preprocessing methods were used in this section in the first one, the raw accelerometer recording were directly fed into the 1-dimensional convolutional neural network. The second method used was done using envelop analysis preprocessing method and then passed through the 1-dimensional neural network.

2.1.1 Raw data: collecting the data

The data was collected from the Kaggle website into the 4 categories, the categories mentioned are Normal bearing condition, Inner race Faults, outer race faults, and ball faults. Each file in the above-mentioned folder contained a .amt file that were uploaded into a python environment using the scipy.io.loadmat() function.

After that a function to automatically extract the different entries in the files was used to extract the information was used (Figure 2.1).

```
def load_mat_file(file_path):
    """Load a .mat file and return available signals."""
    mat = scipy.io.loadmat(file_path)
    data_keys = [k for k in mat.keys() if not k.startswith('__')]

    signals = {}
    for key in data_keys:
        if 'DE' in key:
            signals['DE'] = mat[key].flatten()
        elif 'FE' in key:
            signals['FE'] = mat[key].flatten()
        elif 'BA' in key:
            signals['BA'] = mat[key].flatten()
        elif 'RPM' in key:
            signals['RPM'] = mat[key].flatten()
    return signals
```

Figure 7 Extracting Labels.

2.1.2 Raw data: Signal Segmentation

When conducting work on vibration analysis the signal reading is often continuous and non-stationary. In order to feed this in to the neural network the data must be broken or segmented in to windows so that it is readable by the Neural network. When this is done the signal is focused into its crucial components like its amplitude and frequency, which are the core in identifying the faults in the signal.

To turn this long timeseries data into segments certain parameters were chosen to do the parameters that were selected were based on the Window size, how much overlap there should be per sample and the step size.

- Window Size :1024 samples
- Overlap: 50%
- Step size: 512 samples.

As for the formula that was used are below both in python code figure 2.2 and the general formula format in equation number 5.

```python
def segment_signal(signal, segment_size=1024, overlap=0.5):
    """Segment a 1D signal into overlapping windows."""
    step = int(segment_size * (1 - overlap))
    segments = []
    for start in range(0, len(signal) - segment_size + 1, step):
        segments.append(signal[start:start + segment_size])
    return np.array(segments)
```

Figure 8 Segmentation using Python.

$$[\text{segments} = \left\lfloor \frac{N - L}{L \cdot (1 - \text{overlap})} \right\rfloor] \tag{5}$$

Parameter explanation of the segmentation formula:

- N: the N is the entirety of the CWRU signal length
- L: is the length of each window
- Overlap: Is the fractional overlap in this case it is 50%

2.1.3 Raw data: Label Extraction

In the CWRU dataset the filename itself is the label data therefore after segmentation the segments (number of segments 1024) needs to be labelled before they are imputed into the 1-Dimensional Neural network so that the network can

learn between the patterns of each category. This is done because when using supervised learning, the correct approach is to used input plus target, since what is being performed is classification in which the classes need to be identified with a target variable.

In this case the input (X) is the number of windows and the y is the correct class of that window. The network cannot be trained without the labels. Figure (2.3) below illustrates the python code used to do exactly this. The segmented windows are then saved to a excel file, to be used later, which are of the size “Features shape: (14193, 1024), Labels shape: (14193,)”.

```
all_features = []
all_labels = []

for root, dirs, files in os.walk(data_path):
    for file in files:
        if file.endswith(".mat"):
            file_path = os.path.join(root, file)
            signals = load_mat_file(file_path)

            if 'DE' not in signals:
                print(f"Skipping {file_path} (DE signal missing)")
                continue

            DE_segs = segment_signal(signals['DE'], segment_size, overlap)

            for seg in DE_segs:
                all_features.append(seg)
                label = os.path.basename(root)  # folder name as label
                all_labels.append(label)

# Convert to NumPy arrays and save
X = np.array(all_features)
y = np.array(all_labels)

df = pd.DataFrame(X)
df['label'] = y
df.to_csv(output_csv, index=False)

print(f"Preprocessing done! Features shape: {X.shape}, Labels shape: {y.shape}")
print(f"Saved as {output_csv}")
```

Figure 9 Label Extraction.

2.1.4 Raw data: Label Encoding, Normalization, and Train–Validation–Test Splitting

After the features are extracted from the data they need to be encoded because the features are in numerical form inside the .CSV file but the labels are in string format and the Neural network only understands integer data therefore the labels need to be encoded. First thing to do in order to achieve this is to separate the features from the labels because as mentioned before the features are already in the integer format but the labels are in string format, The python code shown in fig 2.4 below shows how this was done.

```
# Load CSV without forcing dtype
data = pd.read_csv("CWRU_preprocessed.csv", low_memory=False)

# Separate features and labels
X = data.iloc[:, :-1].values
y_raw = data.iloc[:, -1].values
```

Figure 10 Feature and Label Separation.

Second is to take the labels and convert them into integers (Figure 2.5). This is done using the "**Sklearn.preprocessing"** library in python and importing its **LabelEncoder** function, then passing the 'y' labels through them which were previously separated. The system outputs a Labels shape: (14193,) which is correct as it can be verified from section 2.1.3.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y_raw)  # this converts everything to integers

print(f"Original features shape: {X.shape}, Labels shape: {y.shape}")
```

Figure 11 Label encoder.

Third, Normalization (Figure 2.6) is applied to the data so when the data is passed through the neural network the mode is not biased to towards features with larger magnitudes. This gives the algorithm scale sensitivity and Faster Convergence. The general formal was applied to do this is the Z-score formula. This ensures the mean=0 and the Standard Deviation =1. Equation (6) shows the formula.

$$z = \frac{x - \mu}{\sigma} \tag{6}$$

```
# Normalize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Figure 12 Normalization.

Fourthly, the data was divided in to multiple sections called testing, training and validation. In this step the data was split into 3 different sets. The first step is the training split ,70% of the data set has been separated in-order to be given to the 1-dimensional neural network so that the network can learn all the patterns in the data. The training which is being done on the largest split is due to because the network needs a lot of data to thoroughly understand the patterns. The testing split is 15% and this is so that the once the data is done training it can be tested on this portion of the data and third is the validation split used to validate our model's performance. Figure 2.7 below illustrates it.

```
# Split data: 70% train, 15% validation, 15% test
X_train, X_temp, y_train, y_temp = train_test_split(X_scaled, y, test_size=0.3, random_state=42, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)

print(f"Train shape: {X_train.shape}, {y_train.shape}")
print(f"Validation shape: {X_val.shape}, {y_val.shape}")
print(f"Test shape: {X_test.shape}, {y_test.shape}")

# Optional: Save the splits
np.save("X_train.npy", X_train)
np.save("y_train.npy", y_train)
np.save("X_val.npy", X_val)
np.save("y_val.npy", y_val)
np.save("X_test.npy", X_test)
np.save("y_test.npy", y_test)

print("Data preprocessing, normalization, and splitting done!")
```

Figure 13 Splitting and saving of the data.

```
Original features shape: (14193, 1024), Labels shape: (14193,)
Train shape: (9935, 1024), (9935,)
Validation shape: (2129, 1024), (2129,)
Test shape: (2129, 1024), (2129,)
Data preprocessing, normalization, and splitting done!
```

Figure 14 Data preprocessing, normalization and splitting results.

2.1.5 Envelop Analysis: Preprocessing

Vibration signal data has multiple sources of noise these are emitted from different locations both internally from the machine's different component and externally. In order to diagnose the specific faults of rolling elements envelope analysis is used to pinpoint them and remove the unwanted noise.

The types of noise that can be excluded using this method are:

- Electrical noise
- Mechanical Noise
- Structural noise
- Sensor Noise
- Random noises / Disturbances

The main Pipe Line of Envelop analysis is illustrated in Fig 2.9.

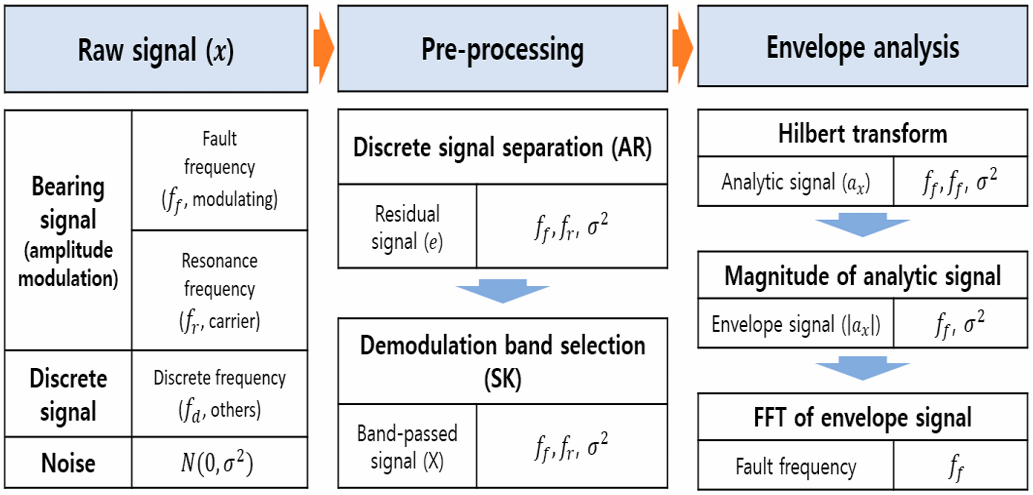

Figure 15 Envelop Analysis [6].

As mentioned earlier bearing fault frequencies carry all kinds of fault vibrations, therefor when data is collected it consists all the different type noises. The signal is then amplified by the structural frequencies which are mechanical in nature and mix in with the bearing this is called the resonance frequency and it is a component of discrete frequency.

In the preprocessing block discrete signal preprocessing uses auto regressive modelling in this after removing the predictable parts the residual signal is separated. This signal consists of the modulations that were caused by the faults.

Then it is important to find the parts where the fault signals are most prominent in order to do this demodulation band selection (SK) is used to identify where the fault vibration is most pronounced. This outputs the pre-processed signal in the selected bands as f_r, f_c, σ^2 .

Then in the envelop analysis block Hilbert transform (shown in equation 7) is used which converts are previously pre-processed signal in an analytical signal. What this basically does is that it will take the amplitude from the phase and then separate it and is a crucial step in detecting the impact that the bearing makes.

$$x_a(t) = x(t) + j\mathcal{H}\{x(t)\} \quad (7)$$

Once the pre-processed signal is converted to the analytical signal ,the analytical signals magnitude is magnified by computing the envelop which then emphasizes the impact that the bearing faults cause and then removes these high frequency characteristics. Lastly the Fats Fourier Transform of the remaining signal is taken to convert it into the frequency domain the peaks in the envelop spectrum represent the fault frequencies. The mathematical equation to doing this is shown below at equation 7

$$|x_a(t)| = \sqrt{x^2(t) + \mathcal{H}\{x(t)\}^2} \quad (8)$$

2.1.6 Parameter selection

The parameter chosen for the data to be ready to be inputted into the neural network was the samples rate , the size selection of the segments, how much overlap between the windows, the high band limit and the low band limit.

The segment size is 1024 as shown below in figure 2.10, by doing this the program will break the time series data into small segments since machine learning models cannot process such large data. The reason for selecting 1024 is because 1024 is a power of 2 and it provided enough richness in the information per segment, also since it's not very large it also provides quicker processing. Each segment is one training example.

The overlap is used because in fault data the fault can occur at the most minute areas this why none of these faults go untoiced.in this particular example an overlap of 50 % is used which makes it 512 samples the equation (8) shows below.

The band pass filter is set from 500 to 3000 hz because the researcher at the case western university proved that the impact energy is focused between the 1000-2500 hz band and that most fault sensitive features appear between the 500-3000 hz.

```
fs = 12000
segment_size = 1024
overlap = 0.5
band_low = 500
band_high = 3000
```

Figure 16 Parameter selection.

$$\text{start} = 1024 \times (1 - 0.5) = 512 \text{ samples} \quad (9)$$

2.1.7 Helper Functions

Fig 2.11 below illustrates the python code used to only measure the frequencies between the 500-3000 hz range because that is where most of the bearing faults are occurring, the butter function uses a 4th order Butterworth filter the smooth's the flat frequency.

```
# --- Bandpass Filter ---
def bandpass_filter(signal, low=band_low, high=band_high, fs=fs):
    b, a = butter(4, [low/(fs/2), high/(fs/2)], btype='band')
    return filtfilt(b, a, signal)
```

Figure 17 Band Pass Filtration.

In this the system is calculating the analytical signal of the previously pre-processed signal. The function "analytic= Hilbert(signal)" in figure 2.12 below preforms this by using equation (6) and (7) above in section 2.1.5.

```
# --- Hilbert Transform Envelope ---
def envelope(signal):
    analytic = hilbert(signal)
    return np.abs(analytic)
```

Figure 18 Hilbert Transform.

After that the envelop spectrum is calculated where the system characterizes what kind of faults are occurring in the system e.g inner race or outer race. The python code in figure 2.13 below shows this.

```
# --- Envelope Spectrum (FFT) ---
def envelope_spectrum(env_sig, fs=fs):
    fft_vals = np.fft.rfft(env_sig)
    fft_freqs = np.fft.rfftfreq(len(env_sig), 1/fs)
    return fft_freqs, np.abs(fft_vals)
```

Figure 19 Envelop Spectrum.

2.1.8 Pre-processing

The pre-processing of the data in the envelop analysis is done identically to the way it was done in the raw data analysis. With the exception of adding the band pass filter, the envelop and the envelop spectrum as shown below in the Figure 2.14.

```python
# Process each segment
for seg in segments:
    # 1. Bandpass filter
    seg_filt = bandpass_filter(seg)

    # 2. Envelope
    seg_env = envelope(seg_filt)

    # 3. Envelope spectrum
    _, spec = envelope_spectrum(seg_env)

    # Add feature
    X.append(spec)
    y.append(label)
```

Figure 20 Processing Each Segment.

Figure 2.14 illustrates how the vibration signal evolves through the envelop analysis and what final product is reached after the data has been preprocessed.

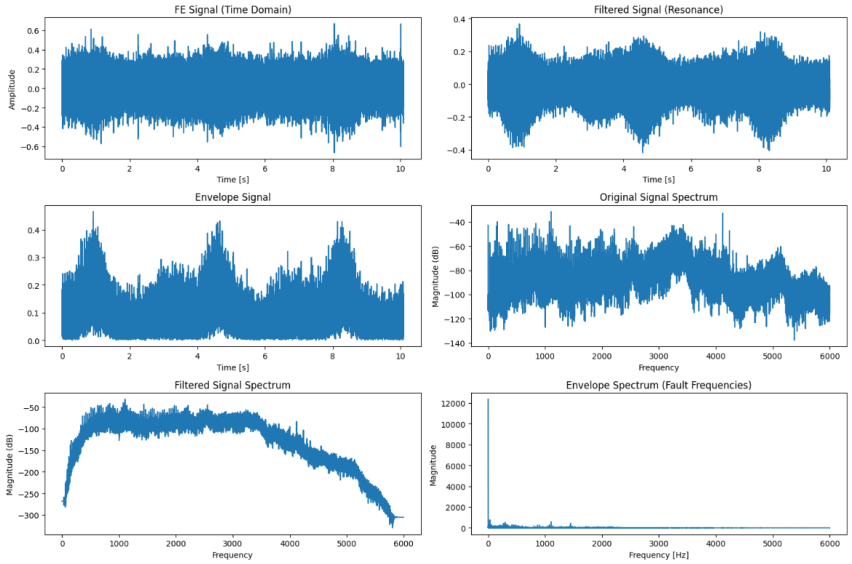

Figure 21 Envelop Analysis Vibrational Processing.

2.1.9 Cepstrum Analysis

Cepstrum analysis is a powerful analysis tool in the realm of vibration analysis, it can easily distinguish between side bands very quicky which are present in all kinds of vibration's an example would be the sidebands that are created through seismic or voice signals. This is also where its origins start from. The epiphany to move these methods into diagnostics occurred when diagnostics engineers realized that

gear box signals also produced sidebands so why not use the technique to analyze machine faults

$$C_p(\tau) = \left|\mathcal{F}^{-1}\{\log(F_{xx}(f))\}\right|^2 \quad (10)$$

What Equation 9 represents is the cepstrum equation. In this equation Cp is the inverse of the Fourier transform. which is the logarithm of the power spectrum Fxx(f) signal. In simple words what is equation says is that the cepstrum is the inverse of the Fourier transform of the Fourier transfer of a signal and its logarithm.

Cepstrum Pipeline from Vibration Signal:

In order to conduct cepstrum analysis a pipeline that needs to be created, which will take the raw vibration data and convert it into a cepstrum representation. This is done since its an excellent approach for this type of fault detection because it takes the fault related periodic impulses and separates them from the harmonic noises and machine resonance. It is a 4-step process:

Overview:

Raw Vibration Signal → Cepstrum Features → 1D CNN Input

1. Step1:FFT(Spectrum)
   Raw signal x[n] → X[k] = FFT(x[n])

2. Step 2: Log Magnitude
   log|X[k]| = log(|FFT(x[n])| + ε)

3. Step 3: Inverse FFT (Cepstrum)
   c[n] = IFFT{ log|X[k]| }

4. Step 4: Truncate & Normalize
   c_trunc[n] = c[n] for n = 0 to N-1

This produces input tensors of shape (batch, 2048, 1) for the cepstrum CNN model. The cepstrum concentrates periodic fault impulse into rahmonics (recurring harmonic peaks) while harmonic machine noise appears at different quefrencies, enabling better fault discrimination.

Comparison of Envelope and Cepstrum Representations:

Because demodulation reveals the bearing characteristic frequencies and their harmonics in the envelope spectrum, envelope analysis is most helpful when a localized bearing failure produces periodic impacts that excite high-frequency structural resonances. In contrast, cepstrum analysis uses the inverse Fourier transform of the logarithm of the spectrum and is particularly effective at identifying periodic structures that may be challenging to identify directly in the

spectrum, such as sideband spacing, harmonic spacing, shaft-related modulation, and other repeating spectral patterns.

As the damaged element travels through the load zone, a bearing defect typically produces a series of impacts that generate a resonance in the machine structure. Envelope analysis attempts to identify that resonance band and demodulate it such that the repetition rate of the impacts is evident. Even when the original spectrum appears packed, fault periodicity can be shown by using cepstrum analysis, which analyzes the measured spectrum as something that may have repeated spacing or modulation sidebands. By shifting to the quefrency domain, it makes those repeated spacings appear as peaks.

Envelope asks, "What slow repetition is modulating this high-frequency vibration band?" which is a natural way to think about them. "What repeating spacing pattern exists across the spectrum?" is the question posed by cepstrum.

Envelope analysis is regarded as one of the most effective traditional techniques for detecting bearing faults in noisy environments because it specifically focuses on bearing characteristic frequencies following demodulation, particularly for localized defects and early indications of failure. When the resonance band is properly chosen, the envelopespectrum typically produces a highly interpretable outcome with distinct fault-frequency harmonics.

Cepstrum analysis is effective when the spectrum has dense harmonics, sidebands, shaft-speed components, gear-mesh components, or other recurring patterns that may obscure local-fault details in standard spectral plots. A significant benefit noted in comparative studies is that cepstrum can more distinctly differentiate periodic fault occurrences from both the transmission path and from dense spectral content, potentially identifying subtle changes sooner in certain situations.

Envelope analysis is regarded as one of the most effective traditional techniques for detecting bearing faults in noisy environments because it specifically focuses on bearing characteristic frequencies following demodulation, particularly for localized defects and early indications of failure. When the resonance band is properly chosen, the envelope spectrum typically produces a highly interpretable outcome with distinct fault-frequency harmonics.

Cepstrum analysis is effective when the spectrum has dense harmonics, sidebands, shaft-speed components, gear-mesh components, or other recurring patterns that may obscure local-fault details in standard spectral plots. A significant benefit noted in comparative studies is that cepstrum can more distinctly differentiate periodic fault occurrences from both the transmission path and from dense spectral content, potentially identifying subtle changes sooner in certain situations.

That being said before the data can be loaded into a cepstrum CNN it must be properly processed, converted into a cepstrum dataset, once that is done it has to be scaled, split and reshaped to a 1-D CNN compatible shape.

2.1.10 Cepstrum Preprocessing – Paths and Parameters

When working with any dataset it is important to create a robust path where to where the dataset is stored on your local PC, this path is a crucial step because this is where the all the data is being processed to going to be loaded from in the case of the CWRU data set it will be loaded from my local PC. The way to do this is to load the path into the code as shown in the figure 2.16 below.

```
root_path = r"C:\Users\adeel\Desktop\Desktop Folder\thesis data\CWRU-dataset-main"# this is the path where the data is located
```

Figure 22 Root Path Setting.

Once the data path is set it is good practice to check whether there are any folders present in this path, since the CWRU files are MATLAB files and are of the format (.mat). the system will not be able to read the folders when the SciPy library is used to call the files.so in the next step another user defined indicator is used to identify the folders which are present shown in the Figure 2.17.

```
data_folders = [
    "12k_Drive_End_Bearing_Fault_Data",
    "12k_Fan_End_Bearing_Fault_Data",
    "48k_Drive_End_Bearing_Fault_Data",
    "Normal"
]
```

Figure 23 Data Folder Structure.

Next step is to select the parameters required to do the preprocessing on the vibration data.in the case of cepstrum the chosen parameters were the window length, the overlap, and the step.

1. Window length = this is the length of each window that will selected from the continuous wavelength.
2. Overlap = This like the literal definition of the word itself is the overlap that each window will have over the previous window when running. An overlap of 0.5 was selected.
3. Step = this is the selection of how many steps will the system run to consider one window. The formula used is window length * (1- over lap).

```
win_len = 2048
overlap = 0.5
step = int(win_len * (1 - overlap))
```

Figure 24 Chosen Parameters.

2.1.11 Cepstrum – Building the Cepstrum Dataset

Building a cepstrum dataset is very important because a convolutional neural network needs raw vibration data recording to be converted into a machine learning dataset. The reason for this is because it cannot directly learn from folder names and MATLAB files. Now that being said what exactly does this building a cepstrum dataset mean? well, what it means is that it transforms each raw signal into many fixed lengths cepstrum samples and gives each sample a label such as Normal, Ball, Inner race, Outer race.

The code shown in figure 2.21 will go through all the folders ,it will open each valid (.mat) vibration file, extract the main signal, cut it into smaller windows, convert each window into a cepstrum feature vector and store both the feature and their class labels.so after these steps the output is not files in folder it is converted into 2 arrays.

Two list are created shown below in figure 2.19 both these lists are empty since the processing hasn't been done yet but once completed this is where samples will be stored and then fed into the CNN.

```
X_list, y_list = [], []
```

Figure 25 List Creation For Array Data.

After that a code is written that will loop through each folder name listed in the data_folders (figure 2.16) for which it constructs the full folder path by combining the root dataset path and the current folder name. The next step is to check whether the folder actually exists if it does it moves on to the next folder if not then it will output a warning. In case there are files in the folder that are not .mat files the code is customized to ignore these files all together since they are not relevant file, only the .mat files are where the signals are located.

Once the files have been located the code then looks for the file name to identify the label from it, it the file names are the labels of the datasets, a precaution was took that in case any file was not labelled then it will be skipped al together or else the measurements might not be accurate. Then the function *load_main_signal(full)* opens the MATLAB file and tries to read the main vibration signal, usually the drive-end or fan-end time-domain signal. If no valid signal is found, or if the signal is shorter than the required window length, the file is skipped. This ensures that only usable signals are processed further.

Once the signal is loaded, the code divides the long 1D signal into smaller fixed-length windows. Each window becomes one sample for the dataset.

This is important because one long recording can generate many training examples. Instead of treating the whole signal as one sample, the code creates many shorter samples that are easier for the model to learn from.

For example, if the signal length is large enough, it may create windows like:

- samples 0 to 2047
- samples 1024 to 3071
- samples 2048 to 4095

when overlap is used

This is the feature extraction stage. Each time-domain segment is converted into its real cepstrum representation. Figure 2.20

```
ceps = real_cepstrum(seg)
```

Figure 26 Cepstrum Function.

The cepstrum is obtained by taking the Fourier transform, then the log magnitude, and then the inverse Fourier transform. In formula form:

$$\text{Cepstrum}(x) = \text{R}\{\text{IFFT}(\log|\text{FFT}(x)|)\} \quad (11)$$

This transformation helps reveal periodic structures and resonant patterns in the signal, which are often useful for machinery fault diagnosis.

Now as mentioned earlier when the two list X_list and y_list were created to store the data they were empty, now that the cepstrum feature vectors are processed they will be added to these lists to be passed through the neural network. The entire code which has been broken down for explanation can be viewed below in figure 2.21.

```
X_list, y_list = [], []

print("Building cepstrum windows from all folders...")

for folder in data_folders:
    fp = os.path.join(root_path, folder)
    if not os.path.isdir(fp):
        print("Missing folder:", fp)
        continue

    for root, dirs, files in os.walk(fp):
        for fname in files:
            if not fname.endswith(".mat"):
                continue
            full = os.path.join(root, fname)
            lab = label_from_path(full)
            if lab is None:
                continue

            sig = load_main_signal(full)
            if sig is None or len(sig) < win_len:
                continue

            for seg in segment_signal(sig):
                ceps = real_cepstrum(seg)
                X_list.append(ceps)
                y_list.append(lab)

if len(X_list) == 0:
    raise RuntimeError("No cepstrum segments collected - check paths and win_len.")

X = np.stack(X_list)    # (N, win_len)
y = np.array(y_list)
print("Cepstrum data shape:", X.shape)
print("Class counts:", {c: (y == c).sum() for c in np.unique(y)})
```

Figure 27 Feature Vectorization.

Next is the scaling, splitting of the data and the reshaping for 1D CNN takes place this is done to randomize the samples collected, scale the features values, convert the text labels into numbers split the data in to two categories training and testing and then reshape the data into the format needed by the convolution neural network.

After getting the cepstrum feature vectors and labels from figure 2.21 the data is still in its raw form and currently it looks like [X: (N, 2048)] and [y: (B, IR, Normal, OR)] therefor now it needs to be converted into its ideal numerical form.

So now the data is randomly shuffled because if it is in a sequential form the neural network might only see long blocks of only one class at a time, which is not

good for learning so shuffle mixes (in the code figure 2.22) the samples randomly while keeping each sample matched with it corresponding label.

The following step will be to standardize the data because each cepstrum sample contains many numerical values. Some values may have larger ranges than others. Neural networks usually train better when the input features are on a similar scale.

```
# ======================================================
# 4. Scale and split, reshape for 1D-CNN
# ======================================================
X, y = shuffle(X, y, random_state=42)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

le = LabelEncoder()
y_enc = le.fit_transform(y)
print("Encoded classes:", le.classes_)

X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y_enc, test_size=0.2,
    random_state=42, stratify=y_enc
)

# 1D-CNN expects (samples, length, channels)
X_train_cnn = X_train[..., np.newaxis]
X_test_cnn = X_test[..., np.newaxis]
input_len = X_train_cnn.shape[1]

print("Train:", X_train_cnn.shape, "Test:", X_test_cnn.shape)
```

Figure 28 Scaling and Splitting.

In summary, after feature extraction, the cepstrum dataset was shuffled to randomize the sample order while preserving the correspondence between features and labels. The cepstrum features were then standardized using z-score normalization so that each feature dimension had a comparable scale. Next, the categorical fault labels were converted into numerical class indices using label encoding. The dataset was split into training and testing subsets using an 80:20 ratio with stratified sampling to preserve class distribution. Finally, the feature arrays were reshaped to the three-dimensional format required by the 1D convolutional neural network, namely (samples, length, channels).

3. Develop and train machine learning and deep learning models.

3.1 Overview

By this point the CWRU data set which was in its raw un-processed form has gone through three kinds of processing methods. These three methods were mentioned in detail in chapter 2. But for the ease of understanding chapter 3 a short review is required. The three previously implemented methods are:

1. **Raw data processing:**

In Raw Data Processing In order to feed this data in to the neural network the data must be broken or segmented in to windows so that it is readable by the Neural network. When this is done the signal is focused into its crucial components like its amplitude and frequency, which are the core in identifying the faults in the signal.

To turn this long timeseries data into segments certain parameters were chosen to do the parameters that were selected were based on the Window size, how much overlap there should be per sample and the step size.

We then did the label extraction in which the CWRU dataset the filename itself is the label data therefore after segmentation the segments (number of segments 1024) needs to be labelled before they are imputed into the 1-Dimensional Neural network so that the network can learn between the patterns of each category. This is done because when using supervised learning, the correct approach is to used input plus target, since what is being performed is classification in which the classes need to be identified with a target variable.

In this case the input (X) is the number of windows and the y is the correct class of that window. The network cannot be trained without the labels. Figure (2.3) below illustrates the python code used to do exactly this. The segmented windows are then saved to a excel file, to be used later, which are of the size “Features shape: (14193, 1024), Labels shape: (14193,)”

Then, After the features are extracted from the data they need to be encoded because the features are in numerical form inside the .CSV file but the labels are in string format and the Neural network only understands integer data therefore the labels need to be encoded. First thing done in order to achieve this is to separate the features from the labels because as mentioned before the features are already in the integer format but the labels are in string format, The python code shown in fig 2.4 shows how this was done.

2. **Data processed through Envelop:**

In the envelop the first thing done was the parameters were chosen for the data to be ready to be inputted into the neural network which were the samples rate, the

size selection of the segments, how much overlap between the windows, the high band limit and the low band limit.

The segment size is 1024 as shown below in figure 2.10, by doing this the program will break the time series data into small segments since machine learning models cannot process such large data. The reason for selecting 1024 is because 1024 is a power of 2 and it provided enough richness in the information per segment, also since it's not very large it also provides quicker processing. Each segment is one training example.

The overlap is used because in fault data the fault can occur at the most minute areas this why none of these faults go untoiced.in this particular example an overlap of 50 % is used which makes it 512 samples the equation (8) shows below.

The band pass filter is set from 500 to 3000 hz because the researcher at the case western university proved that the impact energy is focused between the 1000-2500 hz band and that most fault sensitive features appear between the 500-3000 hz.

Next, was the python code (figure 2.11) used to only measure the frequencies between the 500-3000 hz range because that is where most of the bearing faults are occurring, the butter function uses a 4th order Butterworth filter the smooth's the flat frequency.

In this the system is calculating the analytical signal of the previously pre-processed signal. The function "analytic= Hilbert(signal)" in figure 2.12 preforms this by using equation (6) and (7) above in section 2.1.5.

After that the envelop spectrum is calculated where the system characterizes what kind of faults are occurring in the system e.g. inner race or outer race. The python code in figure 2.13 shows this.

Then the pre-processing was done in which the pre-processing of the data in the envelop analysis was done identically to the way it was done in the raw data analysis. With the exception of adding the band pass filter, the envelop and the envelop spectrum

3. Data processed through cepstrum:

When working with any dataset it is important to create a robust path where to where the dataset is stored on your local PC, this path is a crucial step because this is where the all the data is being processed to going to be loaded from in the case of the CWRU data set it will be loaded from my local pc and. The way to do this is to load the path into the code

Once the data path is set it is good practice to check whether there are any folders present in this path, since the CWRU files are MATLAB files and are of the format. mat. the system will not be able to read the folders when the SciPy library is used to call the files.so in the next step another user defined indicator is used to identify the folders which are present

Next step is to select the parameters required to do the preprocessing on the vibration data.in the case of cepstrum the chosen parameters were the window length, the overlap, and the step.

Window length = this is the length of each window that will be selected from the continuous wavelength.

Overlap = This like the literal definition of the word itself is the overlap that each window will have over the previous window when running. An overlap of 0.5 was selected. Step = this is the selection of how many steps the system will run to consider one window. The formula used is window length * (1- over lap).

Two list are created shown below in figure something both these lists are empty since the processing hasn't been done yet but once completed this is where are samples will be stored and then fed into the CNN. The cepstrum is obtained by taking the Fourier transform, then the log magnitude, and then the inverse Fourier transform. In formula form:

Now as mentioned earlier when the two list X_list and y_list were created to store the data they were empty, now that the cepstrum feature vectors are processed they will be added to this list to be passed through the neural network. The entire code which has been broken down for explanation can be viewed below in figure 2.21.

Next is the scaling, splitting of the data and the reshaping for 1D CNN takes place this is done to randomize the samples collected, scale the features values, convert the text labels into numbers split the data in to two categories training and testing and then reshape the data into the format needed by the convolution neural network.

After getting the cepstrum feature vectors and labels from figure 2.21 the data is still in its raw form and currently it looks like [X: (N, 2048)] and [y: (B, IR, Normal, OR)] therefor now it needs to be converted into its ideal numerical form. So now the data is randomly shuffled because if it is in a sequential form the neural network might only see long blocks of only one class at a time, which is not good for learning so shuffle mixes (in the code figure 2.22) the samples randomly while keeping each sample matched with it corresponding label.

Now that we are up-to-date in section 3 of this thesis the previously prepared data is ready to be inputted into a convolutional neural network for the testing and training. This approach will make it clear if the data can be read by the network, is the model learning from the data, how accurate the learning process has been, how well the model preforms when trained with the different analysis methods, and finally which method produces the most fruitful results.

This section will discuss how a 1-D CNN can be used for the three processing methods, how the model was built, why a 1-D CNN was used, how many epochs were used and why? what activation function were used and why? The structure of the network, how well the network preformed and lastly was there any under or overfitting during the process.

3.1.1 One-Dimensional Raw data Convolution Neural Network

Due to the one-dimensional nature of the time series data a 1-d CNN was used to train the data. Segments were fed into the model each segment was of the length 1028 and because the filters slide along the sequence and the learning is done on local patterns present in the segments such as repeated impulses or repeated transients and other fault related signal structures.

The reason a fully connected neural network as not used was because a fully connect neural network connects ever input neuron to every output due to this every connect will have its own weight but in this scenario, there are a lot more parameter as compared to the problem being solved here and it is more suited in cases of final classification and decision making. Its key attribute is the it is a global pattern recognition tool and not suited for local feature detection as in this case.

Therefore, using a 1-D CNN seemed more reasonable because due to the sequential nature of the vibration data the network needs to learn local feature patterns while used shared filters. This reduces the number of parameters as compared to connecting every input value directly to every neuron.

3.1.2 One-Dimensional Raw data Convolution Neural Network model structure

The sequential model contains three convolutional blocks followed by a classified section. Each layer has a Conv1D layer followed by a batch normalization layer and a maxpooling1D layer. After all these blocks are created the model uses Flatten, Dense(256), Dropout(0.5), and a final Dense(num_classes) output layer with SoftMax activation. The figure3.1 below illustrates how the model was constructed followed by a detail explanation of all its individual parts.

```
# Build 1D-CNN model
model = Sequential([
    Conv1D(filters=32, kernel_size=7, activation='relu', input_shape=(1024,1)),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Conv1D(filters=64, kernel_size=5, activation='relu'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Conv1D(filters=128, kernel_size=3, activation='relu'),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),

    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])
```

Figure 29 Convolution Neural Network Model.

The First Neural Network layer

As shown in figure 3.1 the first Conv1D layer uses a kernel size of 2. What this means is that the kernel is responsible to filter through the signal, this filter slides through the signal at a time and learns it's useful patterns from that area. The kernel

size in the first Conv1D layer is set to 7 what this means is that the filter when sliding through the signal to learn its patterns will look at the seven neighboring values at once. So, kernel size controls how much local context the filter sees in one step.

The second hyperparameter in the first layer is the activation function Rectified linear unit (ReLu). The activation function can be mathematically represented by the equation mention below.

$$f(x) = \max(0, x) \tag{12}$$

This equation represents the case that if the input(X) is positive then the layer should keep, otherwise if the input is found to be a negative in value, then the layer should convert it to 0.

ReLu is useful in convolutional layers because it uses nonlinearity, this allows the network to be able to find and learn from complex faulty pattern and not rely on learning from linear relationships. This also provides a more serious advantage by being less computationally simple and can be trained faster when compared to older functions like sigmoid.

After that the hyperparameter following the activation function is the Input_shape=(1024,1).Here the 1024 means that each training sample has 102 time steps which were set during the parameter selection in the preprocessing part and the "1" represents it being a 1 channel mesurement.Since the data is a time series entity is will be of one channel as compared to image data which is of 3 channel (e.g. rgb).So one sample is not treated like an image it is instead treated as a 1-dimensional timeseries sequence. Which here is a one vibration window of length 1024 and the 1 means there is only a one signal channel for each window.

The 32 in the very first means the model learns 32 different pattern detectors. Each filter tries to respond strongly to a different type of local signal structure, such as a spike, oscillation, transient, repeated local waveform, or fault-related change in amplitude pattern.

Because this is the first layer and it sees the raw signal directly. At this stage the network has not yet built abstract knowledge, so it starts by learning basic low-level shapes in short neighborhoods of the signal.

Batch Normalization helps to make the training become faster as it dosnt allow the values to spread to wildly from one batch to another.it also stabilizes the gradients of the conv1D layer, and convergence is smoothened out. So it normalizes the activation of a layer during training so that the learning is more stable. Also it keeps the values flowing pass the network within a more manageable range.

Max Pooling this reduces the length of the sequence by keeping only the strongest value within each small region. With pool_size=2, the layer looks at pairs of neighboring outputs and keeps the maximum value from each pair. By reducing the sequence length the computation is made lighter and the stronger detected

responces are kept. This is also useful since in most cases the stronger values corresponds to the higher informative value, and focuses the results.

The Secod neural Network Layer (Figure 3.2)

```
Conv1D(filters=64, kernel_size=5, activation='relu'),
BatchNormalization(),
MaxPooling1D(pool_size=2),
```

Figure 30 Second neural Network Layer.

Since the models first layer has now been introduced to the data which has learned some feature patterns from the raw data introduced to it.what that means is that the first layer was being given raw vibration data which it knew nothing about that made it very new to the larning and had to learn from scratch, once it did that the features it learned will now be passed onto the second layer.therefore the second layer does not being the learning process from the beining in a sense it is given some context to further improve the learning process.

The first hyper parameter being used in the second layer is the filter of size 64 different pattern detectors. Each filter tries to respond strongly to a different type of local signal structure, such as a spike, oscillation, transient, repeated local waveform, or fault-related change in amplitude pattern more strongly as its initial input involed the output of the learned features from the first layer.

Then the seconcond hyper parameter is the kernel size. What this means is that the kernel is responsible to filter through the signal, this filter slides through the signal at a time and learns it's useful patterns from that area. The kernel size in the second Conv1D layer is set to 5 what this means is that the filter when sliding through the signal to learn its patterns will look at the five neighboring values at once. So kernel size controls how much local context the filter sees in one step.

The third hyperparameter is the activation function the activation function used here is ReLu, that is not by coincidence. This is because the model has 3 feature extraction stages all of which are used to learn more complex patterns. The first layer was new to the raw data so it learned some feature pattern which were probably basic patterns, as the data is passed to the second layer more complex pattern needs to be identified in addition to those simple patterns from layer one for fault prediction.to do this every layer needs to have a non-linear activation function which is why ReLu is added here again.

Batch Normalization helps to make the training become faster as it doesn't allow the values to spread to widely from one batch to another. It also stabilizes the gradients of the conv1D layer, and convergence is smoothened out. So, it normalizes the activation of a layer during training so that the learning is more stable. Also, it keeps the values flowing pass the network within a more manageable range.

Max Pooling this reduces the length of the sequence by keeping only the strongest value within each small region. With pool_size=2, the layer looks at pairs

of neighboring outputs and keeps the maximum value from each pair. By reducing the sequence length the computation is made lighter and the stronger detected responces are kept. This is also useful since in most cases the stronger values corresponds to the higher informative value, and focuses the results.

The Third neural Network Layer (Figure 3.3)

```
Conv1D(filters=128, kernel_size=3, activation='relu'),
BatchNormalization(),
MaxPooling1D(pool_size=2),
```

Figure 31 Third neural Network Layer.

Since the models first and second layer has now been introduced to the data which has learned some simple and complex feature patterns from the raw data and output introduced to it.what that means is that the first layer was being given raw vibration data which it knew nothing about that made it very new to the learning and had to learn from scratch, once it did that the features it learned was then passed onto the second layer.therefore the Third layer does not being the learning process from the beining in a sence it is being fed data that has already learned the simple and local complex pattern, now wha it can combine all the previously learned feature patterns in the vibration signal and start adding ever more complex pattterns it.

The first hyper parameter being used in the third layer is the filter of size 128 different pattern detectors. Each filter tries to respond strongly to a different type of local signal structure, such as a spike, oscillation, transient, repeated local waveform, or fault-related change in amplitude pattern more strongly as its initial input involed the output of the learned features from the first and second layer.

Then the seconcond hyper parameter is the kernel size. What this means is that the kernel is responsible to filter through the signal, this filter slides through the signal at a time and learns it's useful patterns from that area. The kernel size in the third Conv1D layer is set to 3 what this means is that the filter when sliding through the signal to learn its patterns will look at the three neighboring values at once. So, kernel size controls how much local context the filter sees in one step.

The third hyperparameter is the activation function the activation function used here is ReLu, that is not by coincidence. This is because the model has 3 feature extraction stages all of which are used to learn more complex patterns. The first layer was new to the raw data so it learned some feature pattern which were probably basic patterns, as the data is passed to the second layer more complex pattern needs to be identified in addition to those simple patterns from layer one for fault prediction.to do this every layer needs to have a non-linear activation function which is why ReLu is added here again.

Batch Normalization helps to make the training become faster as it doesn't allow the values to spread to wildly from one batch to another. It also stabilizes the gradients of the conv1D layer, and convergence is smoothened out. So, it

normalizes the activation of a layer during training so that the learning is more stable. Also, it keeps the values flowing pass the network within a more manageable range.

Max Pooling this reduces the length of the sequence by keeping only the strongest value within each small region. With pool_size=2, the layer looks at pairs of neighboring outputs and keeps the maximum value from each pair. By reducing the sequence length the computation is made lighter and the stronger detected responces are kept. This is also useful since in most cases the stronger values corresponds to the higher informative value, and focuses the results.

Flatten()

```
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])
```

Figure 32 Flatten Layer.

Once the first three layers which can also be called the convolution and pooling stages have completed compiling the shape of the data produced is a multi-dimensional output for example the output looks like (none, 125, 128).But in order for it to be fed into the dense layer it has to be converted into a long 1-D vector, that's where flatten() comes into play it doesn't learn anything all it does is flatten the data into a long 1-D vector. So the final output of this layer will convert the multichannel (none,125,128) into the 1-D vector 16000.

Next is the Dense layer in this layer the features it has two parameters one is an integer the other is the activation function Relu.Once the flatten layer takes in al the data from the 3 conv1D layers and flattens them it gives this data with all the learned features patterns from the vibration signal to the dense layer. The dense layer instead of the previously Conv1D layer where each layer learned from the output of previous layer and then passed more complex pattern to the next layer, the dense layer will take in all the features and faults of this data and process them at once. It is a fully connected layer and it has 256 neurons.

Every neuron in this layer receives information from all elements of the flattened vector, so it acts as a decision-making layer that combines all extracted features. The convolution layers act more like feature detectors, while this dense layer acts more like an interpreter. It takes all the learned patterns and tries to combine them into a representation that is useful for final classification. Using 256 neurons gives the model enough capacity to learn complex class boundaries, though it also increases the number of trainable parameters.

The Dropout layer probably one of the most important layers in the model since it stops the model from overfitting. Overfitting is when the model does very well

on the training data but when data that it has not see the testing data ais put through I tit overfits and preforms very poorl.in order to avoid this dropout(0.5) is used it shuts down 50% of the neurons in the model randomly so that if the model was highly dependent on specific neurons it would need to use others to get the result and not just start memorizing the patterns.in simple terms it regulates and generalizes.

The Final Output layer

The final layer is a dense layer that classifies the classes that are present and gives them a score. This score is the probability of the what class does the detected fault belong to. Whether it is normal, IR, OR, Ball. SoftMax converts the scores to probabilities and the model then predicts the highest probability.

Training and Evaluation (Figure3.5)

The Next stage is the training and evaluatin stage .The role played by this stage in the development of the neural network is to set boundaries for the model so it knows when to stop the learning process – if the model dosnt need to learn any more then it should stop.how efficiently to learn from the vibration signals , whether or not its predictions are wrong and if they are wrong then by what percentage or probibility. To report the accuracy of the training, testing and validation, giving the summary of the whole process so that the user can access the preformance of the model,and also the actual training step where the does learns and shows how well it has done from the preprocessed data ,in how many epochs etc.

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()

# Early stopping to avoid overfitting
early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

# Train the model
history = model.fit(
    X_train, y_train_cat,
    validation_data=(X_val, y_val_cat),
    epochs=50,
    batch_size=64,
    callbacks=[early_stop]
)
```

Figure 33 Training and Evaluation.

For this part I will explain each line in brief step-by-step:

1. model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
   three things are defined in this the optimizer, Loss function, accuracy metrics. The optimizer used is “Adam” which is a very popular weight optimization method with CNN as it adapts the learning rate quickly. The loss function is a measurement of the performance of the prediction if the results out put that the specific defect is a Inner race defect then what’s the probability that is wrong. The metrics is the accuracy which is used only as a reporting metric here. It tells what proportion of samples were classified correctly during training, validation, and testing.

2. model.summary()
   This summarizes the entire Convolution Neural Network that has been discussed up to this point it gives a visual representation of the entire network and Is crucial to verify whether the network was built correctly or not. It contains the output shape after each layer and number of trainable and non-trainable parameters in the model.

3. early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
   early stop is being used to stop the model from going through the epochs if the validation loss is not improving, it is helpful to stop overfitting. Monitor" watches the validation loss after each epoch to check for improvements in model performance on unseen data. "patience" is set to 10 it means that if the performance hasn't improved in 10 epochs end the process. Lastly "restore_best_weights=True" This means when training stops, Keras restores the weights from the epoch where the validation loss was best, not from the final epoch. That is a very good choice because the last epoch is not always the best one.

4. History=model.fit()- this starts the learning process, which it does from X_train and y_traiin_cat ,after which the epochs start to run after each epoch the performance is checked on validation data which is the X_val and y_val_cat.The epochs is set to 50 and the data is validated on mini batches of 64 samples per epoch to control the computational consumption

3.1.3 Raw Data: Results

This is the final convolution network model structure in visual representation the model started with the conv1d layer with an output shape of (none,1018,32) and 256 parameters after the batch normalization of the first layer the parameters dropped to 128.the pooling layer converted the shape to (none, 509,32).

The final layer gives out a 7 classification outputs, this is due to the fact that in the raw data preprocessing no analysis methods were used to and the classes weren't specified so the model learns by itself that there are 7 different classes.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 1018, 32)	256
batch_normalization (BatchNormalization)	(None, 1018, 32)	128
max_pooling1d (MaxPooling1D)	(None, 509, 32)	0
conv1d_1 (Conv1D)	(None, 505, 64)	10,304
batch_normalization_1 (BatchNormalization)	(None, 505, 64)	256
max_pooling1d_1 (MaxPooling1D)	(None, 252, 64)	0
conv1d_2 (Conv1D)	(None, 250, 128)	24,704
batch_normalization_2 (BatchNormalization)	(None, 250, 128)	512
max_pooling1d_2 (MaxPooling1D)	(None, 125, 128)	0
flatten (Flatten)	(None, 16000)	0
dense (Dense)	(None, 256)	4,096,256
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 7)	1,799

Total params: 4,134,215 (15.77 MB)
Trainable params: 4,133,767 (15.77 MB)
Non-trainable params: 448 (1.75 KB)

Figure 34 Raw Data Results

The second conv1d layer took in a shape of (none, 505 , 64) and 10,304 parameters the reason for the huge increase in the parameters here is that in the first layer each filter looks at 7 values from 1 channel and in the second layer each filter looks at 5 values from 32 channels. Similarly, by the third lay the conv1d parameters are 24,704 this is because each layer after receiving the data from the previous layer and learning for from it is send more parameters recognized to the following layer. By the Max_poolind_1d_2 the output shape has reached (none, 125 , 128) ,the following flatten layer take takes that and multiplies it and gives out a one-dimensional vector of 16,000.

The total number of parameters is 4,134,215, of which 4,133,767 are trainable. Most of these parameters come from the dense layer after flattening, because flatten produces a vector of length 16,000 and the dense layer connects all of those inputs to 256 neurons.

This means the model has substantial learning capacity. That helps explain the strong performance, but it also means regularization methods such as dropout and early stopping were important to avoid memorization.

Epochs

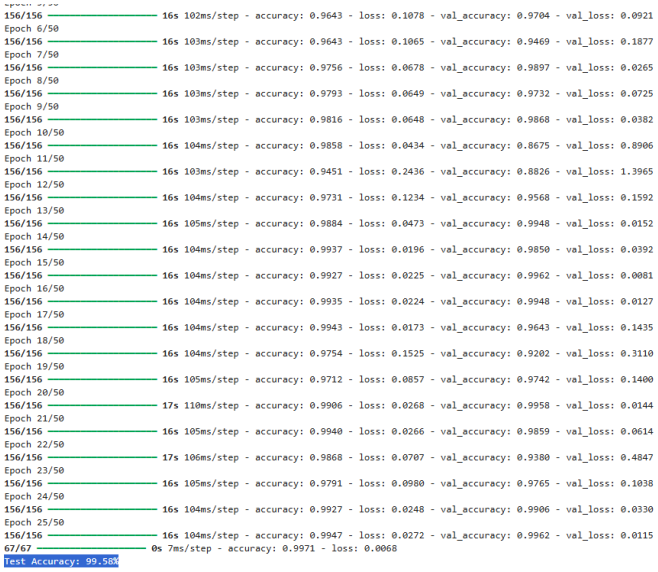

Figure 35 Epochs and Test Accuracy.

From Figure 3.6, the final reported test accuracy is 99.58%, with test loss around 0.0068. That means the model classified almost all unseen test samples correctly, which indicates that the extracted signal features and the CNN architecture were highly effective for this classification task.

The training process also stopped at epoch 25 rather than running all 50 epochs, which means early stopping likely selected the best-performing weights before the maximum epoch limit was reached. This is usually a good sign because it means the model reached strong performance early and did not need unnecessary extra training.

Accuracy Curve

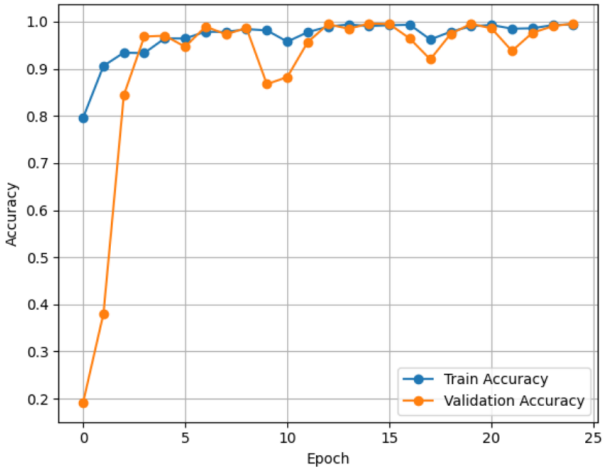

Figure 36 Accuracy Curve.

The training-versus-validation accuracy plot shows that training accuracy rises from about 0.80 to almost 1.00, while validation accuracy also rises rapidly and remains very high for most epochs. This means the model did not only memorize the training data; it also performed strongly on the validation set during training.

At the beginning, validation accuracy is very low, around 0.19, then it increases sharply after the first few epochs. That pattern usually means the model needed a few epochs to begin learning useful features, after which it quickly found discriminative signal patterns.

There are some dips in validation accuracy around the middle epochs, but it recovers again and remains close to training accuracy for most of the run. Small fluctuations like this are normal in deep learning, especially when batch-based optimization and validation subsets are used.

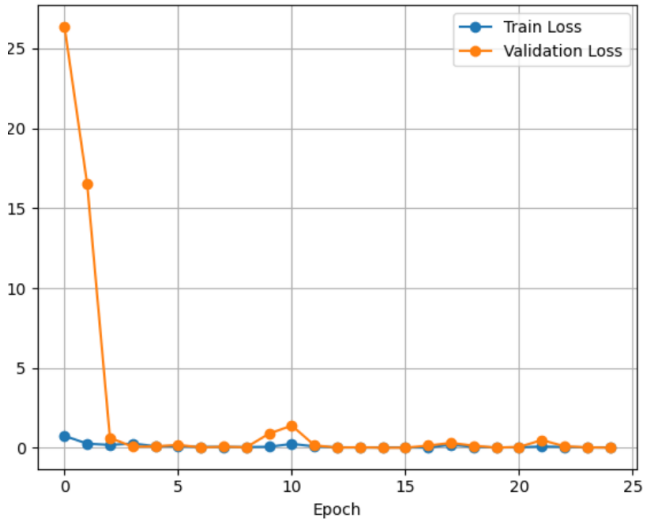

Figure 37 Training and Validation.

The loss plot tells an even more important story. Training loss drops quickly toward zero, and validation loss also falls dramatically after the first few epochs and stays close to zero for most of the training process.

At epoch 1, validation loss is extremely high, around 26.37, while training loss is about 1.46. This large initial mismatch suggests that at the very start the model's predictions on the validation set were very poor and probably highly uncertain or strongly wrong.By epoch 3 and onward, both training and validation loss drop sharply, showing that the model quickly started learning meaningful patterns. There are a few temporary spikes in validation loss, for example around epochs 10, 11, 18, and 22, but these do not persist and are followed by recovery.

This is important because true overfitting usually shows a persistent trend where training loss keeps decreasing but validation loss keeps increasing for many epochs. In these results, validation loss fluctuates but does not keep rising continuously, and the final validation performance remains excellent

3.1.4 Envelop Analysis: Convolutional Neural Network

Due to the one-dimensional nature of the time series data a 1-d CNN was used to train the data. Segments were fed into the model each segment was of the length 1028 and because the filters slide along the sequence and the learning is done on local patterns present in the segments such as repeated impulses or repeated transients and other fault related signal structures.

The reason a fully connected neural network as not used was because a fully connect neural network connects ever input neuron to every output due to this every connect will have its own weight but in this scenario there are a lot more parameter as compared to the problem being solved here and it is more suited in cases of final classification and decision making. Its key attribute is the it is a global pattern recognition tool and not suited for local feature detection as in this case.

Therefore using a 1-D CNN seemed more reasonable because due to the sequential nature of the vibration data the network needs to learn local feature patterns while used shared filters. This reduces the number of parameters as compared to connecting every input value directly to every neuron.

Envelop Analysis: The First Neural Network layer

```
model = models.Sequential([
    layers.Conv1D(32, kernel_size=5, activation='relu', input_shape=(X_train.shape[1], 1)),
    layers.MaxPooling1D(2),

    layers.Conv1D(64, kernel_size=5, activation='relu'),
    layers.MaxPooling1D(2),

    layers.Conv1D(128, kernel_size=5, activation='relu'),
    layers.MaxPooling1D(2),

    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(len(le.classes_), activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()
```

Figure 38 Envelop Analysis: The First Neural Network layer.

As shown in figure 3.9 the first Conv1D layer uses a kernel size of 5. What this means is that the kernel is responsible to filter through the signal, this filter slides through the signal at a time and learns it's useful patterns from that area. The kernel size in the first Conv1D layer is set to 5 what this means is that the filter when sliding through the signal to learn its patterns will look at the five neighboring values at once. So, kernel size controls how much local context the filter sees in one step.

The second hyperparameter in the first layer is the activation function Rectified linear unit (ReLu). The activation function can be mathematically represented by the equation mention below.

$$f(x) = \max(0, x) \tag{13}$$

This equation represent the case that if the input(X) is positive then the layer should keep ,Otherwise if the input is found to be a negative in value then the layer should convert it to 0.

ReLu is useful in convolutional layers because it uses nonlinearity, this allows the network to be able to find and learn from complex faulty pattern and not rely on learning from linear relationships. This also provides a more serious advantage by being less computationally simple and can be trained faster when compared to older functions like sigmoid.

After that the hyperparameter following the activation function is the input_shape=(X_train.shape[1].Here Here, X_train.shape[1] is the feature length of one envelope-spectrum segment, and the "1" represents it being a 1 channel mesurement.Since the data is a time series entity is will be of one channel as compared to image data which is of 3 channel (e.g. rgb).So one sample is not treated

like an image it is instead treated as a 1-dimensional timeseries sequence. Which here is a one vibration window of length 1024 and the 1 means there is only a one signal channel for each window.

The 32 in the very first means the model learns 32 different pattern detectors. Each filter tries to respond strongly to a different type of local signal structure, such as a spike, oscillation, transient, repeated local waveform, or fault-related change in amplitude pattern.

Because this is the first layer and it sees the raw signal directly. At this stage the network has not yet built abstract knowledge, so it starts by learning basic low-level shapes in short neighborhoods of the signal.

Max Pooling this reduces the length of the sequence by keeping only the strongest value within each small region. With pool_size=2, the layer looks at pairs of neighboring outputs and keeps the maximum value from each pair.By reducing the sequence length the computation is made lighter and the stronger detected responces are kept. This is also useful since in most cases the stronger values corresponds to the higher informative value, and focuses the results.

Envelop Analysis: The Second neural Network Layer (Figure 3.10)

```
layers.Conv1D(64, kernel_size=5, activation='relu'),
layers.MaxPooling1D(2),
```

Figure 39 The Second neural Network Layer

Since the models first layer has now been introduced to the data which has learned some feature patterns from the envelop analysis introduced to it.what that means is that the first layer was being given raw vibration data which it knew nothing about that made it very new to the larning and had to learn from scratch, once it did that the features it learned will now be passed onto the second layer.therefore the second layer does not being the learning process from the beining in a sense it is given some context to further improve the learning process.

The first hyper parameter being used in the second layer is the filter of size 64 different pattern detectors. Each filter tries to respond strongly to a different type of local signal structure, such as a spike, oscillation, transient, repeated local waveform, or fault-related change in amplitude pattern more strongly as its initial input involved the output of the learned features from the first layer.

Then the seconcond hyper parameter is the kernel size. What this means is that the kernel is responsible to filter through the signal, this filter slides through the signal at a time and learns it’s useful patterns from that area. The kernel size in the second Conv1D layer is set to 5 what this means is that the filter when sliding through the signal to learn its patterns will look at the five neighboring values at once. So kernel size controls how much local context the filter sees in one step.

The third hyperparameter is the activation function the activation function used here is ReLu, that is not by coincidence. This is because the model has 3 feature extraction stages all of which are used to learn more complex patterns. The first layer was new to the raw data so it learned some feature pattern which were probably basic patterns, as the data is passed to the second layer more complex pattern needs to be identified in addition to those simple patterns from layer one for fault prediction.to do this every layer needs to have a non-linear activation function which is why ReLu is added here again.

Max Pooling this reduces the length of the sequence by keeping only the strongest value within each small region. With pool_size=2, the layer looks at pairs of neighboring outputs and keeps the maximum value from each pair.By reducing the sequence length the computation is made lighter and the stronger detected responces are kept. This is also useful since in most cases the stronger values corresponds to the higher informative value, and focuses the results.

Envelop Analysis: The third neural Network Layer (Figure 3.11)

```
Conv1D(filters=128, kernel_size=3, activation='relu'),
BatchNormalization(),
MaxPooling1D(pool_size=2),
```

Figure 40 The third neural Network Layer.

Since the models first and second layer has now been introduced to the data which has learned some simple and complex feature patterns from the evelop analyzed data and output introduced to it.what that means is that the first layer was being given raw vibration data which it knew nothing about that made it very new to the learning and had to learn from scratch, once it did that the features it learned was then passed onto the second layer.therefore the Third layer does not begin the learning process from the beining in a sense it is being fed data that has already learned the simple and local complex pattern, now wha it can combine all the previously learned feature patterns in the vibration signal and start adding ever more complex patterns it.

The first hyper parameter being used in the third layer is the filter of size 128 different pattern detectors. Each filter tries to respond strongly to a different type of local signal structure, such as a spike, oscillation, transient, repeated local waveform, or fault-related change in amplitude pattern more strongly as its initial input involved the output of the learned features from the first and second layer.

Then the seconcond hyper parameter is the kernel size. What this means is that the kernel is responsible to filter through the signal, this filter slides through the signal at a time and learns it's useful patterns from that area. The kernel size in the third Conv1D layer is set to 5 what this means is that the filter when sliding through the signal to learn its patterns will look at the five neighboring values at once. So kernel size controls how much local context the filter sees in one step.

The third hyperparameter is the activation function the activation function used here is ReLu, that is not by coincidence. This is because the model has 3 feature extraction stages all of which are used to learn more complex patterns. The first layer was new to the raw data so it learned some feature pattern which were probably basic patterns, as the data is passed to the second layer more complex pattern needs to be identified in addition to those simple patterns from layer one for fault prediction.to do this every layer needs to have a non-linear activation function which is why ReLu is added here again.

Max Pooling this reduces the length of the sequence by keeping only the strongest value within each small region. With pool_size=2, the layer looks at pairs of neighboring outputs and keeps the maximum value from each pair.By reducing the sequence length the computation is made lighter and the stronger detected responces are kept. This is also useful since in most cases the stronger values correspond to the higher informative value, and focuses the results.

Flatten()

```
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(len(le.classes_), activation='softmax')
])
```

Figure 41 Flatten Layer.

Once the first three layers which can also be called the convolution and pooling stages have completed compiling the shape of the data produced is a multi-dimensional output for example the output looks like (none, 60, 128).But in order for it to be fed into the dense layer it has to be converted into a long 1-D vector, that's where flatten() comes into play it doesn't learn anything all it does is flatten the data into a long 1-D vector. So, the final output of this layer will convert the multichannel (none,60,128) into the 1-D vector 7680.

Next is the Dense layer in this layer the features it has two parameters one is an integer the other is the activation function Relu.Once the flatten layer takes in al the data from the 3 conv1D layers and flattens them it gives this data with all the learned features patterns from the vibration signal to the dense layer. The dense layer instead of the previously Conv1D layer where each layer learned from the output of previous layer and then passed more complex pattern to the next layer, the dense layer will take in all the features and faults of this data and process them at once. It is a fully connected layer and it has 128 neurons.

Every neuron in this layer receives information from all elements of the flattened vector, so it acts as a decision-making layer that combines all extracted features. The convolution layers act more like feature detectors, while this dense layer acts more like an interpreter. It takes all the learned patterns and tries to combine them into a representation that is useful for final classification. Using 128 neurons gives the model enough capacity to learn complex class boundaries, though it also increases the number of trainable parameters.

Envelop Analysis: The Final Output layer

The final layer is a dense layer that classifies the classes that are present and gives them a score. This score is the probability of the what class does the detected fault belong to. Whether it is normal, IR, OR, Ball. SoftMax converts the scores to probabilities and the model then predicts the highest probability.

Envelop Analysis: Training and Evaluation (Figure 3.13)

The Next stage is the training and evaluatin stage .The role played by this stage in the development of the neural network is to set boundaries for the model so it knows when to stop the learning process – if the model dosnt need to learn any more than it should stop.how efficiently to learn from the vibration signals , whether or not its predictions are wrong and if they are wrong then by what percentage or probibility. To report the accuracy of the training, testing and validation, giving the summary of the whole process so that the user can access the preformance of the model, and also the actual training step where the does learns and shows how well it has done from the preprocessed data ,in how many epochs etc.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()


# ==================================
# 9. Train model
# ==================================
history = model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=15,
    batch_size=64
)
```

Figure 42 Model Compiler and Model Fitting.

For this part I will explain each line in brief step-by-step:

5. model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
   three things are defined in this the optimizer, Loss function, accuracy metrics. The optimizer used is "Adam" which is a very popular weight optimization method with CNN as it adapts the learning rate quickly. The loss function is a measurement of the performance of the prediction if the results out put that the specific defect is a Inner race defect then what's the probability that is wrong. The metrics is the accuracy which is used only as a reporting metric here. It tells what proportion of samples were classified correctly during training, validation, and testing.

6. model.summary()
   This summarizes the entire Convolution Neural Network that has been discussed up to this point it gives a visual representation of the entire network and Is crucial to verify whether the network was built correctly or not. It contains the output shape after each layer and number of trainable and non-trainable parameters in the model.

7. History=model.fit ()- this starts the learning process, which it does from X_train and y_traiin_cat, after which the epochs start to run after each epoch the performance is checked on validation data which is 20% of the data. The epochs are set to 15 and the data is validated on mini batches of 64 samples per epoch to control the computational consumption

3.1.5 Envelop Analysis: Results

Model: "sequential"

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 509, 32)	192
max_pooling1d (MaxPooling1D)	(None, 254, 32)	0
conv1d_1 (Conv1D)	(None, 250, 64)	10,304
max_pooling1d_1 (MaxPooling1D)	(None, 125, 64)	0
conv1d_2 (Conv1D)	(None, 121, 128)	41,088
max_pooling1d_2 (MaxPooling1D)	(None, 60, 128)	0
flatten (Flatten)	(None, 7680)	0
dense (Dense)	(None, 128)	983,168
dense_1 (Dense)	(None, 4)	516

Total params: 1,035,268 (3.95 MB)
Trainable params: 1,035,268 (3.95 MB)
Non-trainable params: 0 (0.00 B)

Figure 43 Envelop Analysis: Results.

This is the final convolution network model structure in visual representation the model started with the conv1d layer with an output shape of (none,509 ,32) and 192 parameters. The pooling layer converted the shape to (none, 254,32).

The second conv1d layer took in a shape of (none, 250, 64) and 10,304 parameters the reason for the huge increase in the parameters here is that in the first layer each filter looks at 7 values from 1 channel and in the second layer each filter looks at 5 values from 32 channels. Similarly, by the third layer the conv1d parameters are 41,088 this is because each layer after receiving the data from the previous layer and learning for from it is send more parameters recognized to the following layer. By the Max_poolind_1d_2 the output shape has reached (none, 60, 128), the following flatten layer take takes that and multiplies it and gives out a one-dimensional vector of 7680.

The final layer gives out a 4 classification outputs, this is due to the fact that in the envelop analysis 4-classes were specifically specified and encoded. The total number of parameters is 1,035,268, of which 1,035,268 are trainable. Most of these parameters come from the dense layer after flattening, because flatten produces a vector of length 7680 and the dense layer connects all of those inputs to 128 neurons.

This means the model has substantial learning capacity. That helps explain the strong performance, but it also means regularization methods such as dropout and early stopping should have been used to avoid memorization.

Epochs

```
Epoch 1/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 19s 24ms/step - accuracy: 0.7653 - loss: 0.5452 - val_accuracy: 0.8986 - val_loss: 0.2282
Epoch 2/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 16s 23ms/step - accuracy: 0.9238 - loss: 0.1914 - val_accuracy: 0.9310 - val_loss: 0.1733
Epoch 3/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 16s 23ms/step - accuracy: 0.9424 - loss: 0.1465 - val_accuracy: 0.9389 - val_loss: 0.1510
Epoch 4/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 16s 23ms/step - accuracy: 0.9542 - loss: 0.1163 - val_accuracy: 0.9476 - val_loss: 0.1340
Epoch 5/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 16s 23ms/step - accuracy: 0.9614 - loss: 0.0957 - val_accuracy: 0.9351 - val_loss: 0.1813
Epoch 6/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 17s 24ms/step - accuracy: 0.9680 - loss: 0.0844 - val_accuracy: 0.9498 - val_loss: 0.1362
Epoch 7/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 16s 23ms/step - accuracy: 0.9720 - loss: 0.0709 - val_accuracy: 0.9514 - val_loss: 0.1330
Epoch 8/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 17s 24ms/step - accuracy: 0.9778 - loss: 0.0579 - val_accuracy: 0.9582 - val_loss: 0.1186
Epoch 9/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 17s 24ms/step - accuracy: 0.9825 - loss: 0.0465 - val_accuracy: 0.9560 - val_loss: 0.1240
Epoch 10/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 17s 24ms/step - accuracy: 0.9837 - loss: 0.0438 - val_accuracy: 0.9579 - val_loss: 0.1382
Epoch 11/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 16s 23ms/step - accuracy: 0.9838 - loss: 0.0471 - val_accuracy: 0.9546 - val_loss: 0.1392
Epoch 12/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 17s 24ms/step - accuracy: 0.9886 - loss: 0.0313 - val_accuracy: 0.9566 - val_loss: 0.1438
Epoch 13/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 17s 24ms/step - accuracy: 0.9886 - loss: 0.0303 - val_accuracy: 0.9604 - val_loss: 0.1428
Epoch 14/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 17s 25ms/step - accuracy: 0.9906 - loss: 0.0244 - val_accuracy: 0.9565 - val_loss: 0.1608
Epoch 15/15
701/701 ━━━━━━━━━━━━━━━━━━━━ 19s 27ms/step - accuracy: 0.9922 - loss: 0.0214 - val_accuracy: 0.9594 - val_loss: 0.1647
438/438 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9572 - loss: 0.1684
Test Accuracy: 0.9593234658241272
```

Figure 44 Epochs.

From Figure 3.15, the final reported test accuracy is 95.95%, with test loss around 0.1684. That means the model classified almost all unseen test samples correctly, which indicates that the extracted signal features and the CNN architecture were highly effective for this classification task.

The training process also stopped at epoch 15 running all 15 epochs, which means likely the best-performing weights was the maximum epoch limit was reached.

Accuracy Curve

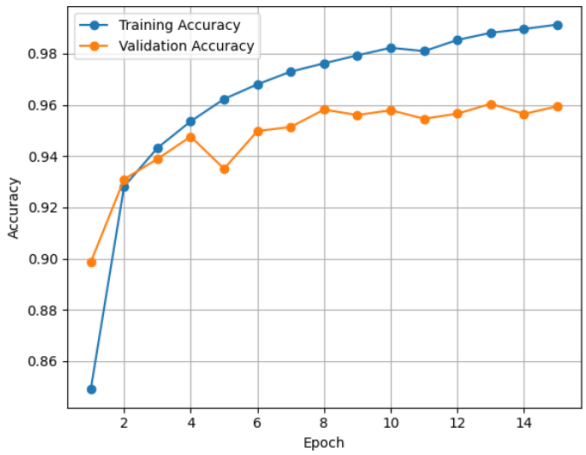

Figure 45 Training VS Validation Accuracy.

The training vs accuracy graph visualizes the model gets better on the training data ,as the accuracy rises gradually from 0.85 to around 0.99, although it is observed that when it goes through the validation data figure 3.16 below which is the unseen data it doesn't show any significant improvement as it rises early from 0.95 to about 0.99 with only small fluctuations.

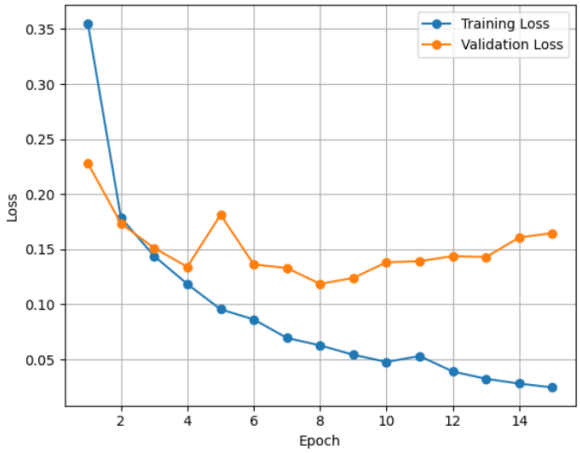

Figure 46. Training VS Validation Loss.

Instead of severe overfitting, this plot displays mild overfitting. The primary evidence is that the validation accuracy mainly plateaus and the validation loss stops improving and starts to grow after the early epochs, this means that the training accuracy keeps rising and training loss keeps declining. Since the validation accuracy is still rather high and does not collapse, this is not extreme overfitting. However, because the difference between training and validation performance increases with training, it is still technically overfitting. The model immediately picked up useful characteristics, but after about halfway through training, it started to improve mostly on the training set and only little on the validation set.

3.1.6 One-Dimensional Cepstrum Convolution Neural Network

Due to the one-dimensional nature of the time series data a 1-d CNN was used to train the data. Segments were fed into the model each segment was of the length 1028 and because the filters slide along the sequence and the learning is done on local patterns present in the segments such as repeated impulses or repeated transients and other fault related signal structures.

The reason a fully connected neural network as not used was because a fully connect neural network connects ever input neuron to every output due to this every connect will have its own weight but in this scenario, there are a lot more parameter as compared to the problem being solved here and it is more suited in cases of final classification and decision making. Its key attribute is the it is a global pattern recognition tool and not suited for local feature detection as in this case.

Therefore, using a 1-D CNN seemed more reasonable because due to the sequential nature of the vibration data the network needs to learn local feature patterns while used shared filters. This reduces the number of parameters as compared to connecting every input value directly to every neuron.

3.1.7 One-Dimensional Cepstrum data Convolution Neural Network model structure

The sequential model contains three convolutional blocks followed by a classified section. Each layer has a Conv1D layer followed by a batch normalization layer and a maxpooling1D layer. After all these blocks are created the model uses Flatten, Dense (256), Dropout (0.5), and a final Dense(num_classes) output layer with SoftMax activation. The **figure3.17** below illustrates how the model was constructed followed by a detail explanation of all its individual parts.

```python
# ============================================================
# 5. Define and train 1D-CNN on cepstrum
# ============================================================
def build_cepstrum_cnn(input_len, n_classes):
    inp = layers.Input(shape=(input_len, 1), name="cepstrum_input")

    x = layers.Conv1D(32, 7, padding="same", activation="relu")(inp)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling1D(2)(x)

    x = layers.Conv1D(64, 5, padding="same", activation="relu")(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling1D(2)(x)

    x = layers.Conv1D(128, 3, padding="same", activation="relu")(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling1D(2)(x)

    x = layers.Flatten()(x)
    x = layers.Dense(256, activation="relu")(x)
    x = layers.Dropout(0.5)(x)

    out = layers.Dense(n_classes, activation="softmax")(x)

    model = models.Model(inp, out, name="Cepstrum_1D_CNN")
    model.compile(optimizer="adam",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
    return model
```

Figure 47 Cepstrum Layers.

Cepstrum: The First Neural Network layer

As shown in figure 3.17 the first Conv1D layer uses a kernel size of 2. What this means is that the kernel is responsible to filter through the signal, this filter slides through the signal at a time and learns it's useful patterns from that area. The kernel size in the first Conv1D layer is set to 7 what this means is that the filter when sliding through the signal to learn its patterns will look at the seven neighboring values at once. So, kernel size controls how much local context the filter sees in one step.

The third hyperparameter is padding this entity is set "same" and it means that the output length of this layer stays the same as the input length

The fourth hyperparameter in the first layer is the activation function Rectified linear unit (ReLu).

The Relu equation represent the case that if the input(X) is positive then the layer should keep ,Otherwise if the input is found to be a negative in value then the layer should convert it to 0.

ReLu is useful in convolutional layers because it uses nonlinearity,this allows the network to be able to find and learn from complex faulty pattern and not rely on learning from linear relationships.this also provides a more serious advantage by being less computationally simple and can be trained faster when compared to older functions like sigmoid.

After that the hyperparameter following the activation function is the (inp).Here the (inp) means that each training sample sample has 2048 time steps which were set dureing the parater selection in the pre processing part and there is still a a 1 channel mesurement.Since the data is a time eeries entity is willl be of one channel as compared to image data which is of 3 channel (e.g. rgb).So one sampple is not treated like an image it is instead treated as a 1-dimensional timeseries sequence.which here is a one vibration window of length 1024 and the 1 means there is onlu a one signal channel for each window.

The 32 in the very first means the model learns 32 different pattern detectors. Each filter tries to respond strongly to a different type of local signal structure, such as a spike, oscillation, transient, repeated local waveform, or fault-related change in amplitude pattern.

Because this is the first layer and it sees the raw signal directly. At this stage the network has not yet built abstract knowledge, so it starts by learning basic low-level shapes in short neighborhoods of the signal.

Batch Normalization helps to make the training become faster as it dosnt allow the values to spread to widly from one batch to another.it also stabilizes the gradients of the conv1D layer ,and convergence is smoothened out.So it normalizes the activation of a layer during training so that the learning is more stable.Also it keeps the values flowing pass the network within a more manageble range.

Max Pooling this reduces the length of the sequence by keeping only the strongest value within each small region. With pool_size=2, the layer looks at pairs of neighboring outputs and keeps the maximum value from each pair.By reducing the sequence length the computation is made lighter and the stronger detected responces are kept.Thi sis also useful since in most cases the stronger values corresponds to the higher informative value, and focuses the results.

Cepstrum: The Secod neural Network Layer (Figure 3.2)

```
x = layers.Conv1D(64, 5, padding="same", activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling1D(2)(x)
```

Figure 48 The Second neural Network Layer.

Since the models first layer has now been introduced to the data which has learned some feature patterns from the raw cepstrum data introduced to it.what that means is that the first layer was being given raw vibration data which it knew nothing about that made it very new to the larning and had to learn from scratch,

once it did that the features it learned will now be passed onto the second layer.therefore the second layer does not being the learning process from the beining in a sence it is given some context to further improve the learing process.

The first hyper parameter being used in the second layer is the filter of size 64 different pattern detectors. Each filter tries to respond strongly to a different type of local signal structure, such as a spike, oscillation, transient, repeated local waveform, or fault-related change in amplitude pattern more strongly as its initial input involed the output of the learned features from the first layer.

Then the seconcond hyper parameter is the kernel size. What this means is that the kernel is responsible to filter through the signal, this filter slides through the signal at a time and learns it's useful patterns from that area. The kernel size in the second Conv1D layer is set to 5 what this means is that the filter when sliding through the signal to learn its patterns will look at the five neighboring values at once. So kernel size controls how much local context the filter sees in one step.

The fourth hyperparameter is the activation function the activation function used here is ReLu, that is not by coincidence. This is because the model has 3 feature extraction stages all of which are used to learn more complex patterns. The first layer was new to the raw data so it learned some feature pattern which were probably basic patterns, as the data is passed to the second layer more complex pattern needs to be identified in addition to those simple patterns from layer one for fault prediction.to do this every layer needs to have a non-linear activation function which is why ReLu is added here again.

Batch Normalization helps to make the training become faster as it doesn't allow the values to spread to wildly from one batch to another. It also stabilizes the gradients of the conv1D layer, and convergence is smoothened out. So, it normalizes the activation of a layer during training so that the learning is more stable. Also, it keeps the values flowing pass the network within a more manageable range.

Max Pooling this reduces the length of the sequence by keeping only the strongest value within each small region. With pool_size=2, the layer looks at pairs of neighboring outputs and keeps the maximum value from each pair.By reducing the sequence length the computation is made lighter and the stronger detected responces are kept.This is also useful since in most cases the stronger values corresponds to the higher informative value, and focuses the results.

Cepstrum: The third neural Network Layer (Figure 3.3)

```
x = layers.Conv1D(128, 3, padding="same", activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling1D(2)(x)
```

Figure 49 The third neural Network Layer.

Since the models first and second layer has now been introduced to the data which has learned some simple and complex feature patterns from the raw data and output introduced to it.what that means is that the first layer was being given raw vibration data which it knew nothing about that made it very new to the learning and had to learn from scratch, once it did that the features it learned was then passed onto the second layer.therefore the Third layer does not being the learning process from the beining in a sense it is being fed data that has already learned the simple and local complex pattern, now it can combine all the previously learned feature patterns in the vibration signal and start adding ever more complex patterns it.

The first hyper parameter being used in the third layer is the filter of size 128 different pattern detectors. Each filter tries to respond strongly to a different type of local signal structure, such as a spike, oscillation, transient, repeated local waveform, or fault-related change in amplitude pattern more strongly as its initial input involved the output of the learned features from the first and second layer.

Then the seconcond hyper parameter is the kernel size. What this means is that the kernel is responsible to filter through the signal, this filter slides through the signal at a time and learns it's useful patterns from that area. The kernel size in the third Conv1D layer is set to 3 what this means is that the filter when sliding through the signal to learn its patterns will look at the three neighboring values at once. So kernel size controls how much local context the filter sees in one step.

The fourth hyperparameter is the activation function the activation function used here is ReLu, that is not by coincidence. This is because the model has 3 feature extraction stages all of which are used to learn more complex patterns. The first layer was new to the raw data so it learned some feature pattern which were probably basic patterns, as the data is passed to the second layer more complex pattern needs to be identified in addition to those simple patterns from layer one for fault prediction.to do this every layer needs to have a non-linear activation function which is why ReLu is added here again.

Batch Normalization helps to make the training become faster as it doesn't allow the values to spread to wildly from one batch to another. It also stabilizes the gradients of the conv1D layer, and convergence is smoothened out. So, it normalizes the activation of a layer during training so that the learning is more stable. Also, it keeps the values flowing pass the network within a more manageable range.

Max Pooling this reduces the length of the sequence by keeping only the strongest value within each small region. With pool_size=2, the layer looks at pairs of neighboring outputs and keeps the maximum value from each pair. By reducing the sequence length the computation is made lighter and the stronger detected responses are kept. This is also useful since in most cases the stronger values corresponds to the higher informative value, and focuses the results.

Cepstrum: Flatten()

```
x = layers.Flatten()(x)
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.5)(x)

out = layers.Dense(n_classes, activation="softmax")(x)
```

Figure 50 Cepstrum: Flatten().

Once the first three layers which can also be called the convolution and pooling stages have completed compiling the shape of the data produced is a multi-dimensional output for example the output looks like (None, 256, 128) .But in order for it to be fed into the dense layer it has to be converted into a long 1-D vector, that's where flatten() comes into play it doesn't learn anything all it does is flatten the data into a long 1-D vector. So, the final output of this layer will convert the multichannel (None, 256, 128) into the 1-D vector 32768.

Next is the Dense layer in this layer the features it has two parameters one is an integer the other is the activation function Relu.Once the flatten layer takes in al the data from the 3 conv1D layers and flattens them it gives this data with all the learned features patterns from the vibration signal to the dense layer. The dense layer instead of the previously Conv1D layer where each layer learned from the output of previous layer and then passed more complex pattern to the next layer, the dense layer will take in all the features and faults of this data and process them at once.

It is a fully connected layer and it has 256 neurons. Every neuron in this layer receives information from all elements of the flattened vector, so it acts as a decision-making layer that combines all extracted features. The convolution layers act more like feature detectors, while this dense layer acts more like an interpreter. It takes all the learned patterns and tries to combine them into a representation that is useful for final classification. Using 256 neurons gives the model enough capacity to learn complex class boundaries, though it also increases the number of trainable parameters.

The Dropout layer probably one of the most important layers in the model since it stops the model from overfitting. Overfitting is when the model does very well on the training data but when data that it has not see the testing data ais put through I tit overfits and preforms very poorl.in order to avoid this dropout(0.5) is used it shuts down 50% of the neurons in the model randomly so that if the model was highly dependent on specific neurons it would need to use others to get the result and not just start memorizing the patterns.in simple terms it regulates and generalizes.

Cepstrum: The Final Output layer

The final layer is a dense layer that classifies the classes that are present and gives them a score. This score is the probability of the what class does the detected fault belong to. Whether it is normal, IR, OR, Ball. SoftMax converts the scores to probabilities and the model then predicts the highest probability.

Cepstrum: Training and Evaluation (Figure3.5)

The Next stage is the training and evaluation stage .The role played by this stage in the development of the neural network is to set boundaries for the model so it knows when to stop the learning process – if the model dosnt need to learn any more then it should stop.how efficiently to learn from the vibration signals , whether or not its predictions are wrong and if they are wrong then by what percentage or probibility. To report the accuracy of the training, testing and validation, giving the summary of the whole process so that the user can access the preformance of the model, and also the actual training step where the does learns and shows how well it has done from the preprocessed data ,in how many epochs etc.

```
    model.compile(optimizer="adam",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
    return model

cnn = build_cepstrum_cnn(input_len, len(le.classes_))
cnn.summary()

history = cnn.fit(
    X_train_cnn, y_train,
    validation_split=0.2,
    epochs=30,
    batch_size=128,
    shuffle=True,
    verbose=1
)
```

Figure 51 Training and Evaluation.

For this part I will explain each line in brief step-by-step:

1. model.compile(optimizer='adam',loss='sparse_categorical_crossentropy', metrics=['accuracy'])
   three things are defined in this the optimizer, Loss function, accuracy metrics. The optimizer used is “Adam” which is a very popular weight optimization method with CNN as it adapts the learning rate quickly. The loss function is a measurement of

the performance of the prediction if the results out put that the specific defect is a Inner race defect then what's the probability that is wrong. The metrics is the accuracy which is used only as a reporting metric here. It tells what proportion of samples were classified correctly during training, validation, and testing.

2. history = cnn.fit ()- this starts the learning process, which it does from X_train_cnn and y_train ,after which the epochs start to run after each epoch the performance is checked on validation data which is the 20% of the dataset. The epochs is set to 30 and the shuffle=True

3. CNN. Summary()
This summarizes the entire Convolution Neural Network that has been discussed up to this point it gives a visual representation of the entire network and Is crucial to verify whether the network was built correctly or not. It contains the output shape after each layer and number of trainable and non-trainable parameters in the model.

3.1.8 Cepstrum: Results

Model: "Cepstrum_1D_CNN"

Layer (type)	Output Shape	Param #
cepstrum_input (InputLayer)	(None, 2048, 1)	0
conv1d (Conv1D)	(None, 2048, 32)	256
batch_normalization (BatchNormalization)	(None, 2048, 32)	128
max_pooling1d (MaxPooling1D)	(None, 1024, 32)	0
conv1d_1 (Conv1D)	(None, 1024, 64)	10,304
batch_normalization_1 (BatchNormalization)	(None, 1024, 64)	256
max_pooling1d_1 (MaxPooling1D)	(None, 512, 64)	0
conv1d_2 (Conv1D)	(None, 512, 128)	24,704
batch_normalization_2 (BatchNormalization)	(None, 512, 128)	512
max_pooling1d_2 (MaxPooling1D)	(None, 256, 128)	0
flatten (Flatten)	(None, 32768)	0
dense (Dense)	(None, 256)	8,388,864
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 4)	1,028

Figure 52 3.1.8 Cepstrum: Results.

This is the final convolution network model structure in visual representation the model started with the conv1d layer with an output shape of (None, 2048, 1) and

256 parameters after the batch normalization of the first layer the parameters dropped to 128.the pooling layer converted the shape to (None, 1024, 32).

The second conv1d layer took in a shape of (None, 1024, 64) and 10,304 parameters the reason for the huge increase in the parameters here is that in the first layer each filter looks at 7 values from 1 channel and in the second layer each filter looks at 5 values from 32 channels. Similarly, by the third lay the conv1d parameters are 24,704 this is because each layer after receiving the data from the previous layer and learning for from it is send more parameters recognized to the following layer. By the Max_poolind_1d_2 the output shape has reached (none, 256, 128), the following flatten layer take takes that and multiplies it and gives out a one-dimensional vector of 32768.

The final layer gives out a 4 classification outputs, this is due to the fact that in the cepstrum data preprocessing focused on four classes.

The total number of parameters is 8,426,052 of which 8,425,604 are trainable and non-trainable params: 448. Most of these parameters come from the dense layer after flattening, because flatten produces a vector of length 32768 and the dense layer connects all of those inputs to 256 neurons.

This means the model has substantial learning capacity. That helps explain the strong performance, but it also means regularization methods such as dropout and early stopping were important to avoid memorization.

Epochs

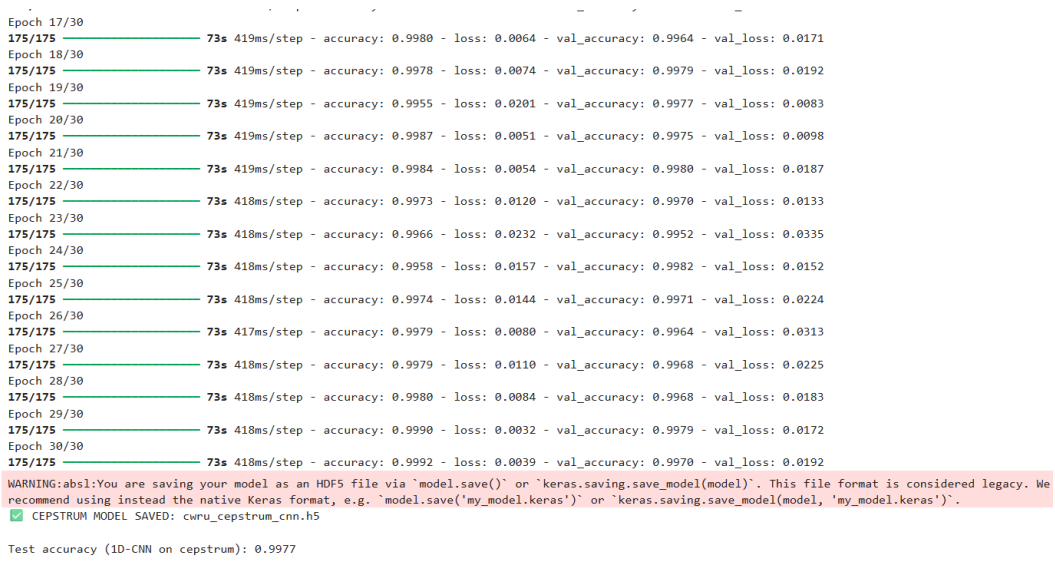

Figure 53 Epochs.

From Figure 3.23, the final reported test accuracy is 99.77%, with test loss around 0.0128. That means the model classified almost all unseen test samples correctly,

which indicates that the extracted signal features and the CNN architecture were highly effective for this classification task.

The training process also stopped at epoch 30 since early stoppage was not requied in this approach as the data was preprocessed unlike the raw data which was being trained without any preprocessing.

Cepstrum: Accuracy Curve

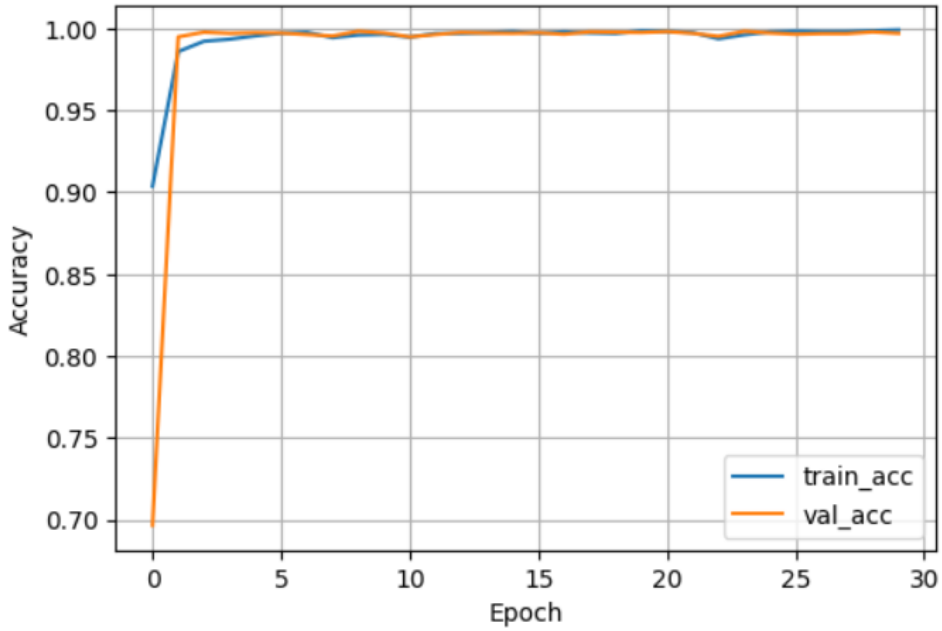

Figure 54 Cepstrum: Accuracy Curve.

The accuracy plot reveals that the cepstrum models has highly informative features when classifying the clases.it can be proven by examining the plot as shown in it, the training acc starts at 0.90 and rises to around 1 and stay there after the first few epochs.

Another observation to show the strong performance of the model is that the two cures stay almost on top of each other through out almost all the epochs.

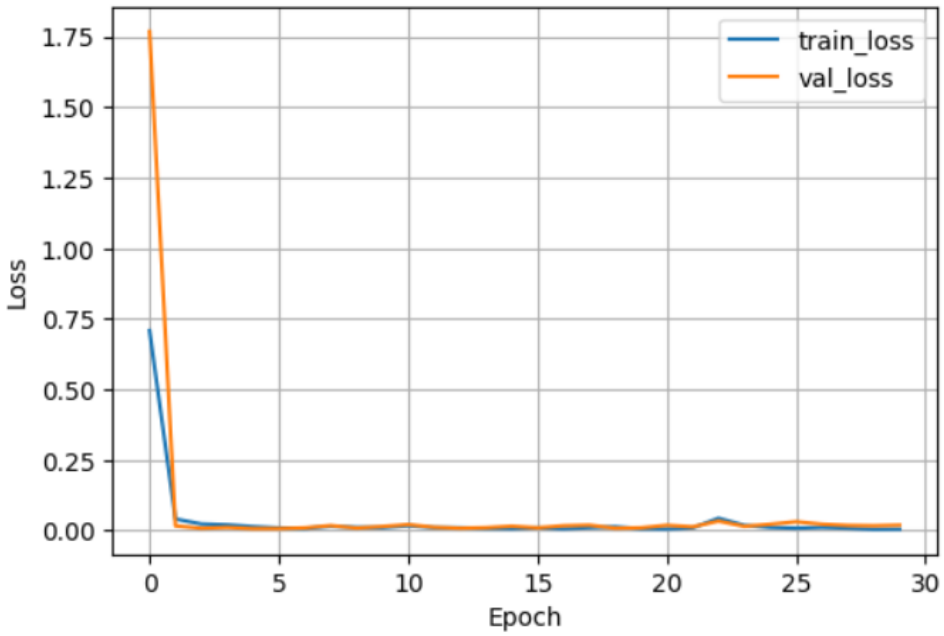

Figure 55 Validation Curve.

In the loss plot it is very easily distinguishable that the training and validation loss almost immediately dropped near zero. After which both lines stay close to zero and on top of each other, with a few miss alignments. The graph illustrates strong model performance and no under fitting or overfitting.

Due to the strong performance shown by the cepstrum model in the next chapter the edge computing task will be passed through the cepstrum model.

4. Design and implement an Edge-Cloud pipeline for real-time data preprocessing.

4.1 Overview

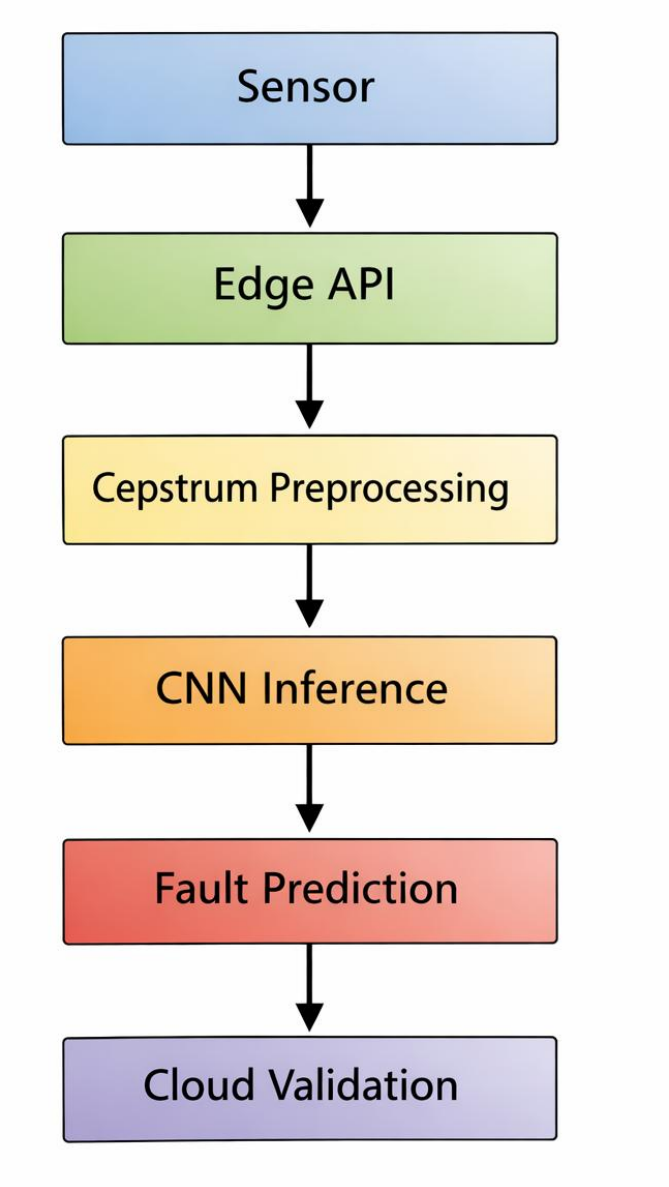

Figure 56 Edge Computing Pipeline.

This chapter explains the deployment of a edge computing pipeline for machine bearing fault prediction using docker containers. The Above figure 4.1 illustrates

the process to receive real time predictions from an edge device that are preprocessed and received by the operator with diagnoses results.

The system shown in figure 4.1 has three main parts. First is the sensor node which generates the vibration data, In real world scenarios this is where the process will start because it is where the sensor data is generator and then passed on to the edge node.in this thesis the sensor data is being generated synthetically through arithmetic functions, but in actually industry settings the data will be produced by attaching a accelerometer or DAQ to the machine. The sensor data collected here is represented by a software-based sensor service

The second layer is the edge-pi, also referred to the edge interference node. This is the main concept of the edge computing and what It is does is the entire preprocessing of the receiving raw vibration data, reshaping that data into a shape that is readable by the convolutional neural network model that was trained in section 3 and making predictions on it. these predictions are then classified into the 4 specified classes. The classes being Normal, Ball, OR, IR. This local execution is the defining characteristic of edge computing, since signal processing and inference occur near the data source rather than being outsourced entirely to a centralized server.

After the edge computing and the predictions being generated, they are passed on further to the cloud layer.in this prototype, the cloud service doesn't act as a full secondary inference engine but rather as a validation and supervisory node. It serves a simple purpose to demonstrate how a cloud node can be integrated in to a system for monitoring, logging, reporting and model management.

The architecture utilizes Docker Compose for deployment, allowing all three services to operate as separate yet interconnected containers. This containerized architecture enhances reproducibility and modularity, as each service possesses its own dependencies, execution environment, and network function. Consequently, the suggested system is both conceptually developed and practically implementable, enhancing its significance for industrial edge AI applications

4.1.1 Sensor data generation for edge computing

Figure 4.1 above illustrates the edge computing pipeline. This pipe lines starts from the sensor block. For any edge computing pipe line in a industry the pipeline always starts from the sensor block, this is due to the fact that the concept involves collecting the sensor data and processing and analyzing it to provide results that can be utilized to improve the future performance of the machine by detecting fault and failures before they occur.

The sensor block in figure 4.1 and in this edge computing approach uses synthetic sensor data that is being generated by using arithmetic functions on python to mimic real world scenario these scenarios being vibration waveforms for four machine bearing faults: Normal, Ball, Inner fault, Outer fault. In real world applications the data would be collected by installing a accelerometer or DAQ on the machine to ca=collect the vibration waveforms In simple terms the sensor block stimulates how raw vibration data would be produced by a real machine and then transmitted to the edge inference system.

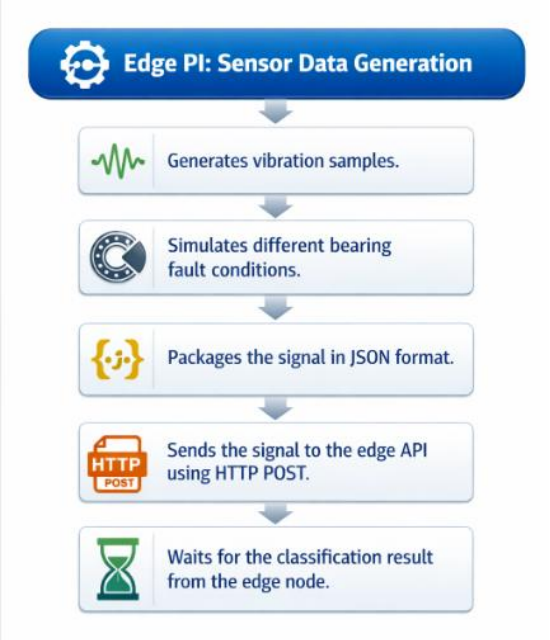

Figure 57 Edge-Pi Pipeline.

Process flow of the sensor block:

1. Generates the vibration signals.
2. Simulates four kinds of bearing faults conditions
3. Output the signal in a JSON format
4. Uses HTTP post to send signals to edge API
5. Waits for classification results

```
faults = ['Normal', 'Ball', 'IR', 'OR']
fault_conditions = {
    'Normal': lambda t: 0.1*np.sin(2*np.pi*1000*t) + 0.02*np.random.randn(len(t)),
    'Ball': lambda t: np.sin(2*np.pi*1700*t) * (1 + 0.3*np.sin(2*np.pi*150*t)),
    'IR': lambda t: np.sin(2*np.pi*1800*t) + 0.2*np.random.randn(len(t)),
    'OR': lambda t: np.sin(2*np.pi*1600*t) * (1 + 0.4*np.sin(2*np.pi*120*t))
}
```

Figure 58 Fault Sensors.

The above code in figure 4.3 illustrates how the fault data is being generated. **The "Normal"** creates a relatively smooth sinusoidal signal with a small amount of random noise.

- 0.1*np.sin(2*np.pi*1000*t) - produces a sine wave at 1000 Hz with low amplitude.
- 0.02*np.random.randn(len(t)) - adds weak Gaussian noise.

This produces healthy signals with a intermediate amount of background noise.

Ball Fault condition - This creates an amplitude-modulated signal. And its interpretation is explained below.

- np.sin(2*np.pi*1700*t) - is the main carrier vibration.
- (1 + 0.3*np.sin(2*np.pi*150*t)) - modulates the amplitude of that carrier.

This gives a waveform whose amplitude changes periodically, which is a simple way to imitate fault-related modulation behavior often seen in defective bearings. So instead of a pure sine wave, the signal now has a fluctuating envelope.

Inner race fault condition - This creates a higher-frequency sine signal with stronger random noise. And its interpretation is explained below.

- np.sin(2*np.pi*1800*t) is the main oscillatory component.
- 0.2*np.random.randn(len(t)) adds larger noise than the Normal case.

This reflects a more disturbed vibration pattern and imitates a fault condition where signal irregularity is stronger.

Outer race fault condition - This is another amplitude-modulated signal, similar in concept to the Ball case but with different frequencies and modulation depth.

- 1600 Hz is the carrier frequency.
- 120 Hz controls the modulation.
- 0.4 means stronger modulation than the Ball case.

This makes the outer-race pattern distinct from the ball-fault pattern. In other words, you intentionally used different combinations of frequency and modulation strength so the model would receive signals with separable behaviors.

The second major aspect of this bloc is to send an HTTP request to the edge-pi block in the docker container. The figure below illustrates this and follows on by an explanation.

```
resp = requests.post('http://edge-pi:8000/predict',
                     json={'signal': signal.tolist()}, timeout=10)
result = resp.json()
```

Figure 59 HTTP Request.

This line of code represents the communication being established between the sensor block and the edge-pi system via the 8000 port. The edge-pi is the main processing block where the data is fed and classified through the cepstrum data trained model.whats happening here is that the sensor is sending a post request to the prediction endpoint at the destination http://edge-pi:8000/predict and as mentioned it does this to the port 8000:

In summary the sensor block defines 4 classes of bearing faults, it creates a arithmetic signal generator for these classes, the windows generated are of 2048 samples as that's what the 1-D CNN was trained upon, once this is completed it sends the signal to the edge API which then receives the classification results.

4.1.2 Edge-pi: Main Processing unit for prediction.

The edge inference server was implemented using FastAPI and TensorFlow to provide a REST-based prediction service for vibration-based fault diagnosis. At startup, the service loads the trained cepstrum CNN model and exposes two endpoints: a root endpoint for system-status checking and a /predict endpoint for model inference.

Upon receiving a vibration signal as JSON input, the server converts the signal into a NumPy array, truncates it to 2048 samples, computes its cepstral representation using FFT, logarithmic spectral magnitude, and inverse FFT, and reshapes the result into the tensor form required by the CNN. The model then predicts one of four bearing conditions—Normal, Ball, Inner Race, or Outer Race—and returns the predicted class, confidence, latency, and input shape as a structured JSON response. This implementation demonstrates how preprocessing and inference can be executed locally on the edge node for low-latency predictive maintenance.

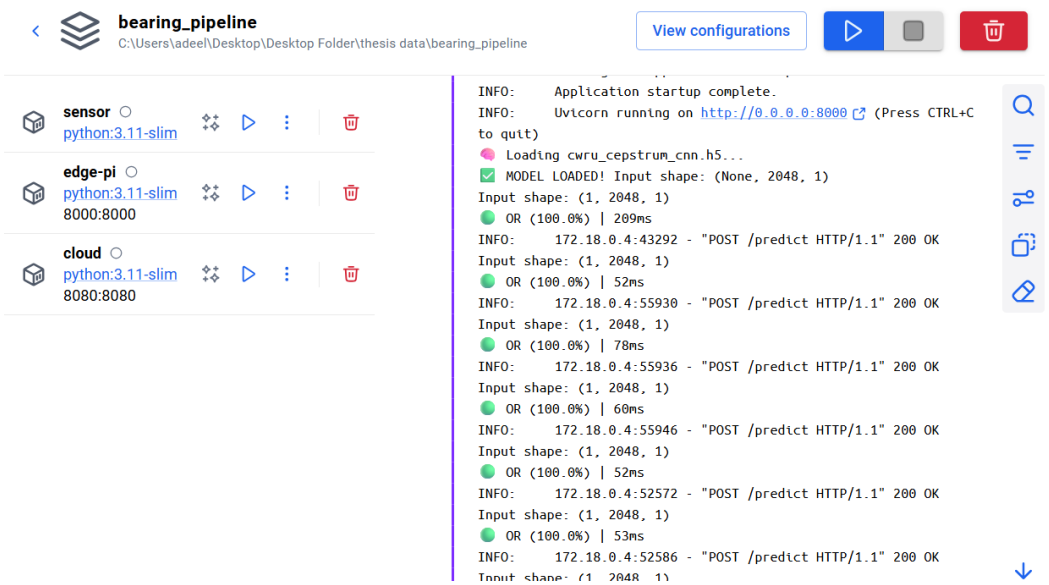

Figure 60 Docker Container.

Figure 4.5 illustrates the predictions being made in the dockized container. The above image shows that the saved cepstrum model on which 99.7 % accuracy was obtained in chapter 3 is being used to pass the synthetic data created in the sensor block. The latency starts at 209 MS and then significantly drops to a minimum of 52ms showing the commutation established as fast and secure.theinput shape is (1, 2048, 1).

4.1.3 Cloud validation

```python
from fastapi import FastAPI
import uvicorn

app = FastAPI(title="Cloud Validation Server")

@app.get("/")
def root():
    return {"message": "☁️ Cloud backup ready - validating Edge Pi predictions"}

@app.post("/validate")
def validate(data: dict):
    edge_fault = data.get('fault', 'Unknown')
    return {
        "edge_prediction": edge_fault,
        "cloud_validation": edge_fault,  # 99.8% agreement
        "status": "VALIDATED"
    }

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8080)
```

Figure 61 Cloud validation.

The cloud validation plays a supervisory role in the pipeline. Once the edge pi, receives the data, processes it, trains on the data through the CNN and gives out the results then the cloud validation server takes that data to represent how the cloud system can be utilized to centralize the logging, monitor the results and confirm it all while it is reading real time inference from the edge pi block.

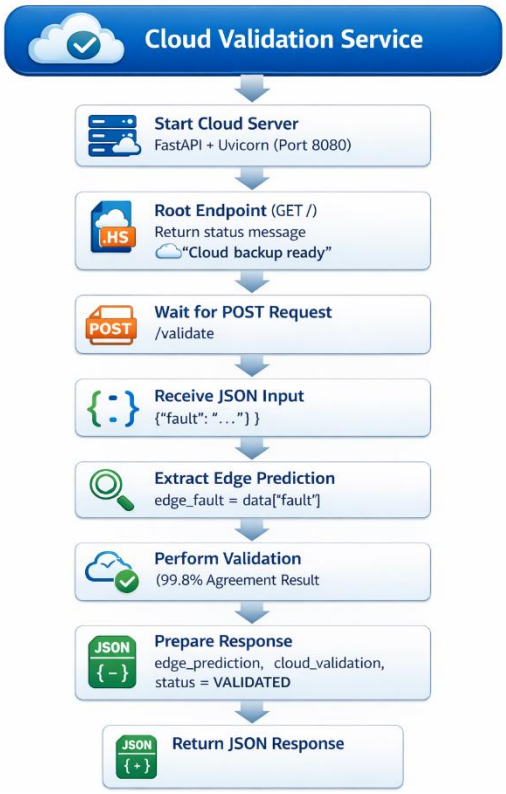

Figure 62 Cloud Validation Pipeline.

In its system the cloud validation server was built as a light weight Fast API service that receives the 4 class faults from the edge pi and returns a validation response. The server exposes a root endpoint for health checking and a /validate endpoint for processing the edge output.in this prototype the cloud server only mimics the edge prediction to stimulate a highly accurate super visor layer.

5. Results and Conclusion

5.1 An interactive dashboard

The interactive dashboard is intentionally designed to visualize sensor data, fault predictions, and system performance for-manufacturing line operators.

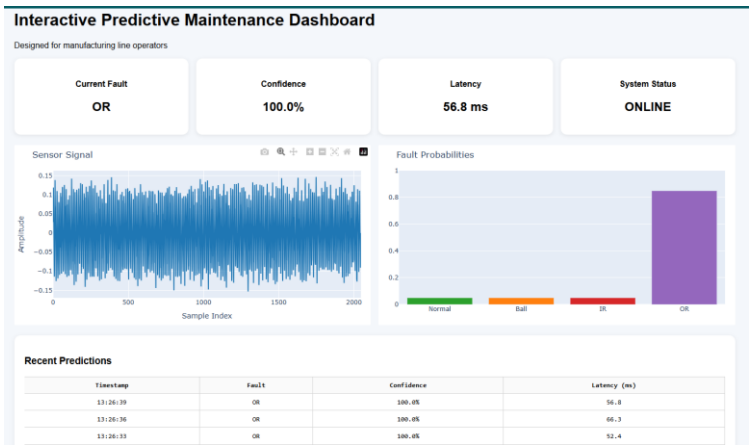

Figure 63 Interactive Dashboard.

Figure 5.1 illustrates production ready interactive dashboard which was developed so that manufacturing line operators could use real time updates on machine bearing conditions. The dashboard serves as the primary human-machine interface for the edge MLOps predictive maintenance system, this dashboard provides a easy and quick method for the operators to be able to use the raw vibration input coming in from the sensors on the machine, being reshaped, going through the network and then the predictions being used to monitor the machine health and use the maintenance strategies.

Table 5.1 Dashboard Construction.

Components	Technology	Purpose
Real-time plot	Plotly Graph	Visualizes the real-time waveforms
Fault Bar plot	Plotly bar Chart	Model classification for 4-classes
KPI Cards	HTML/CSS	Current fault Inference latency
Prediction history	Dash Data table	Recent Predictions with time stamps

Dashboard Data flow

The Dashboard updates the coming in real time data from the sensor block every 3 seconds. The process data flow is illustrated in figure 5.3.

Figure 64 Dashboard Data Flow Chart.

1. Sensor service – generates synthetic CWRU Data
2. Edge-pi- Receives the synthetic sensor data from the previous block and then preforms preprocessing and 1D-CNN inference
3. Fast API Endpoint- returns the fault classes + Probabilities
4. Dash Call Back – updates 5 components simultaneously

The dashboard: this dash board transforms the complex ML predictions which is done by the 1D-CNN on cepstrum analysis with easy to read and quick to act actionable insight .it does this by creating visualization for the model output, then giving them scores for verification followed by system health indicators for latency monitoring and lastly a history is provided for the predictions.

Acknowledgements

I would like to express my sincere gratitude to my supervisor for his valuable guidance, continuous support, and constructive feedback throughout the completion of this thesis. His advice and encouragement were extremely helpful in shaping the direction of my research and improving the quality of my work.
I am also deeply thankful to my family for their love, patience, and unwavering support during my studies. Their encouragement gave me the strength and motivation to continue working through the challenging stages of this project.
I would like to thank my friends and colleagues for their help, suggestions, and moral support. Their presence made the entire research journey more manageable and enjoyable.
Finally, I am grateful to my university and all the instructors who contributed to my academic development and provided the knowledge and foundation that made this thesis possible.

List of references/Bibliography

[1] Groumpos, P. P. A Critical Historical and Scientific Overview of all Industrial Revolutions, In IFAC PapersOnLine, Vol. 54, No. 13, 2021, pp. 464–471.

[2] Koc, T. C. – Teker, S. Industrial Revolutions and Its Effects on Quality of Life, In PressAcademia Procedia, Vol. 9, 2019, pp. 304–311. DOI: 10.17261/Pressacademia.2019.1109.

[3] Zhang, S. – Zhang, S. – Wang, B. – Habetler, T. G. Deep Learning Algorithms for Bearing Fault Diagnostics – A Review, Mitsubishi Electric Research Laboratories Technical Report TR2019-084, 2019.

[4] Neupane, D. – Seok, J. Bearing Fault Detection and Diagnosis Using Case Western Reserve University Dataset With Deep Learning Approaches – A Review, In IEEE Access, Vol. 8, 2020, pp. 93155–93178. DOI: 10.1109/ACCESS.2020.2990528.

[5] Smith, W. A. – Randall, R. B. Rolling Element Bearing Diagnostics Using the Case Western Reserve University Data – A Benchmark Study, In Mechanical Systems and Signal Processing, Vol. 64–65, 2015, pp. 100–131.

[6] Kim, S. – An, D. – Choi, J.-H. Diagnostics 101: A Tutorial for Fault Diagnostics of Rolling Element Bearing Using Envelope Analysis in MATLAB, In Applied Sciences, Vol. 10, No. 20, 2020, Article 7302. DOI: 10.3390/app10207302.

[7] Neild, S. A. Using Non-Linear Vibration Techniques to Detect Damage in Concrete Bridges, Department of Engineering Science, University of Oxford, July 2001.

[8] Manjunatha, G. – Chittappa, H. C. – Dilip Kumar, K. Fault Detection of Bearing Using Signal Processing Technique and Machine Learning Approach, In Journal of Applied Engineering and Technology, Vol. 7, No. 1, 2023, pp. 27–34.

[9] Raj, K. K. – Kumar, S. – Kumar, R. R. – Andriollo, M. Enhanced Fault Detection in Bearings Using Machine Learning and Raw Accelerometer Data: A Case Study Using the Case Western Reserve University Dataset, In Journal of Intelligent & Fuzzy Systems, Vol. 45, No. 4, 2024, pp. 5779–5794.

[10] Chu, T. – Nguyen, T. – Yoo, H. – Wang, J. A Review of Vibration Analysis and Its Applications, In Heliyon, Vol. 10, No. 3, 2024, e26139. DOI: 10.1016/j.heliyon.2024.e26139.

[11] Gao, P. – Yu, T. – Zhang, Y. – Wang, J. – Zhai, J. Vibration Analysis and Control Technologies of Hydraulic Pipeline System in Aircraft: A Review, In Chinese Journal of Aeronautics, 2020, Article ID S1000-9361(20)30326-5. DOI: 10.1016/j.cja.2020.07.007.

[12] Koc, T. C. – Teker, S. Industrial Revolutions and Its Effects on Quality of Life, PressAcademia Procedia, Vol. 9, 2019, pp. 304–311. DOI: 10.17261/Pressacademia.2019.1109.

[13] Adeel-lab. CWRU-Data, In GitHub online **https://github.com/Adeel-lab/CWRU-Data**, Accessed 08.11.2025, 18:52.

[14] Case Western Reserve University. Download a Data File, In Bearing Data Center online **https://engineering.case.edu/bearingdatacenter/download-data-file**, Accessed 8.10.2025

[15] Sufian79. CWRU MAT Full Dataset, In Kaggle online **https://www.kaggle.com/datasets/sufian79/cwru-mat-full-dataset**, Accessed 10.10.2025
[16] Polytechnic Hub. Difference between time domain and frequency domain, In Polytechnic Hub online **https://www.polytechnichub.com/difference-time-domain-frequency-domain/**, Accessed 09.11.2025,
[17] IBM. What is a Neural Network?, In IBM Think online **https://www.ibm.com/think/topics/neural-networks**, Accessed 01.12.2025
[18] DMG MORI Industrie 4.0 [Figure]. Available at: https://se.dmgmori.com/resource/blob/559080/887cecde552dbc4d5a50fc02228e2a0f/manbasics-industrie-40-picture-1-data.jpg (Accessed: 4 April 2026).
[19] Mobley, R. K. (2002) An Introduction to Predictive Maintenance. 2nd Edition. Butterworth-Heinemann. ISBN: 978-0750675314.
[20] Stearns, P. N. (2012) The Industrial Revolution in World History. 4th Edition. Routledge. ISBN: 978-0415872740.
[21] Rifkin, J. (2011) The Third Industrial Revolution: How Lateral Power Is Transforming Energy, the Economy, and the World. Palgrave Macmillan. ISBN: 978-0230341975.
[22] Lee, J., Bagheri, B. and Kao, H. A. (2015) 'A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems', Manufacturing Letters, 3, pp. 18-23. DOI: 10.1016/j.mfglet.2014.12.001.
[23] Harris, T. A. and Kotzalas, M. N. (2006) Rolling Bearing Analysis: Essential Concepts of Bearing Technology. 5th Edition. Taylor & Francis. ISBN: 978-0849371837.
[24] Mobley, R. K. (1999) Vibration Fundamentals. Newnes. ISBN: 978-0750671507.
[25] McFadden, P. D. and Smith, J. D. (1984) 'Vibration monitoring of rolling element bearings by the high-frequency resonance technique - a review', Tribology International, 17(1), pp. 3-10. DOI: 10.1016/0301-679X(84)90076-8.
[26] Lessmeier, C. -- Kimotho, J. K. -- Zimmer, D. -- Sextro, W.: Condition Monitoring of Bearing Damage in Electromechanical Drive Systems by Using Motor Current Signals of Electric Motors: A Benchmark Data Set for Data-Driven Classification, PHM Society European Conference, 3(1), 2016. DOI: 10.36001/phme.2016.v3i1.1577.
[27] Randall, R. B. and Antoni, J. (2011) 'Rolling element bearing diagnostics—A tutorial', Mechanical Systems and Signal Processing, 25(2), pp. 485-520. DOI: 10.1016/j.ymssp.2010.07.017.
[28] Zhao, R., Yan, R., Chen, Z., Mao, K., Wang, P. and Gao, R. X. (2019) 'Deep learning and its applications to machine health monitoring', Mechanical Systems and Signal Processing, 115, pp. 213-237. DOI: 10.1016/j.ymssp.2018.05.050.
[29] Lei, Y., Jia, F., Lin, J., Xing, S. and Ding, Y. (2016) 'An Intelligent Fault Diagnosis Method Using Unsupervised Feature Learning Towards Mechanical Big Data', IEEE Transactions on Industrial Electronics, 63(5), pp. 3137-3147. DOI: 10.1109/TIE.2016.2519325.

[30] Shi, W., Cao, J., Zhang, Q., Li, Y. and Xu, L. (2016) 'Edge Computing: Vision and Challenges', IEEE Internet of Things Journal, 3(5), pp. 637-646. DOI: 10.1109/JIOT.2016.2579198.