MICROCONTROLLER-BASED REAL-TIME MOTOR BEARING FAULT DETECTION AND DIAGNOSIS USING 1D CONVOLUTIONAL NEURAL NETWORKS

SERTAÇ KILIÇKAYA

Master's Thesis

Graduate School
Izmir University of Economics
Izmir
2022

MICROCONTROLLER-BASED REAL-TIME MOTOR BEARING FAULT DETECTION AND DIAGNOSIS USING 1D CONVOLUTIONAL NEURAL NETWORKS

SERTAÇ KILIÇKAYA

A Thesis Submitted to

The Graduate School of Izmir University of Economics

Master Program in Electrical and Electronics Engineering

Izmir

2022

ABSTRACT

MICROCONTROLLER-BASED REAL-TIME MOTOR BEARING FAULT DETECTION AND DIAGNOSIS USING 1D CONVOLUTIONAL NEURAL NETWORKS

Kılıçkaya, Sertaç

Master Program in Electrical and Electronics Engineering

Advisor: Prof. Dr. Türker İnce

January, 2022

Continuous machine monitoring provides a real-time intelligence on the status and health of the machinery; hence it is a very common practice that avoids unexpected machine failures in the industry. One of the most common causes of rotating machine failures are bearing faults, and early detection of bearing defects allows replacement of faulty bearing rather than the motor itself. Therefore, the lifetime and condition of electric motor bearings are of great interest to end users to sustain continuous plant operation. Traditional bearing fault detection systems perform classification using hand-crafted features; hence they require significant computational cost, avoiding real-time applications. On the other hand, 1D Self-Organized Operational Neural Networks (1D Self-ONNs) and its special case 1D Convolutional Neural Networks (1D CNNs)

are the promising alternatives that encapsulates feature extraction and classification phases into a single learning body, thus allowing more efficient systems in terms of computational complexity. In this study, first, the effectiveness of 1D Self-ONNs and CNNs for bearing fault diagnosis is shown on two benchmark datasets. In addition, using an on-board accelerometer, several minutes of 3-axis accelerometer data is collected from two different single-phase induction motors with four different bearing health conditions on the motor test setup at Izmir University of Economics. A 1D CNN model is then trained, quantized, and deployed to Arm Cortex-M4 based microcontroller to evaluate the bearing fault diagnosis performance in real-world scenario. The experimental results indicate that it is feasible to detect and classify bearing faults in real-time on low-power microcontrollers using 1D CNNs.

Keywords: Embedded Machine Learning, Self-Organized Operational Neural Networks (Self-ONNs), Convolutional Neural Networks (CNNs), Condition-Based Monitoring (CBM), Rolling Element Bearings (REBs), Bearing Fault Detection and Diagnosis (BFDD).

ÖZET

1B EVRİŞİMSEL SİNİR AĞLARI İLE MİKRODENETLEYİCİ TABANLI GERÇEK ZAMANLI MOTOR RULMAN ARIZASI TESPİTİ VE TEŞHİSİ

Kılıçkaya, Sertaç

Elektrik-Elektronik Mühendisliği Tezli Yüksek Lisans Programı

Tez Danışmanı: Prof. Dr. Türker İnce

Ocak, 2022

Sürekli makine durum izlemesi, makinelerin durumu ve sağlığı hakkında gerçek zamanlı bilgi sağlaması nedeniyle endüstride beklenmedik makine arızalarını önleyen çok yaygın bir uygulamadır. Dönen makine arızalarının en yaygın nedenlerinden biri rulman arızalarıdır ve rulman arızalarının erken tespiti, motorun kendisinden ziyade arızalı rulmanın değiştirilmesini sağlar. Bu nedenle, elektrik motor rulmanlarının ömrü ve durumu, endüstriyel tesislerin kesintisiz çalışmasını sürdürmek için son kullanıcılar açısından büyük önem taşımaktadır. Geleneksel rulman arıza tespit sistemleri, manuel öznitelikler çıkararak sınıflandırma gerçekleştirir ve yüksek işlem gereksinimi sebebiyle gerçek zamanlı uygulamayı zorlaştırırlar. Öte yandan, 1B Operasyonel Sinir

Ağları (1B OSA) ve bunların özel bir durumu olan 1B Evrişimsel Sinir Ağları (1B ESA), otomatik öznitelik çıkarma ve sınıflandırma aşamalarını tek bir öğrenme gövdesinde toplayan daha az işlem gerektiren verimli alternatiflerdir. Bu çalışmada, ilk olarak, 1B OSA'ların ve ESA'ların rulman arıza teşhisindeki etkinliği iki açık kaynak veri seti kullanılarak gösterilmiştir. Ayrıca, İzmir Ekonomi Üniversitesi'ndeki motor test düzeneği kullanılarak iki çeşit tek fazlı asenkron motordan dört farklı rulman sağlığı koşulu için birkaç dakikalık 3 eksen ivmeölçer verisi toplanmıştır. Toplanan veri kullanılarak, bir 1B ESA modeli eğitilip, model katsayıları nicemlendikten sonra Arm Cortex-M4 tabanlı mikrodenetleyiciye yüklenmiştir ve bu sayede gerçek bir motor düzeneğinde modelin rulman arıza teşhis performansı gözlemlenmiştir. Deneysel sonuçlar, 1B ESA'lar kullanılarak düşük güçlü mikrodenetleyiciler ile rulman hatalarının gerçek zamanlı tespit ve teşhisinin mümkün olduğunu göstermektedir.

Anahtar Kelimeler: Gömülü Makine Öğrenmesi, Operasyonel Sinir Ağları (OSA), Evrişimsel Sinir Ağları (ESA), Durum Bazlı Bakım, Bilyalı Rulmanlar, Rulman Arızası Tespit ve Teşhisi.

To my family…

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Türker İnce for introducing me to the field of embedded machine learning, for inspiring me to select that field as the subject of my master's thesis, and for guiding me throughout this study.

I would also like to thank Prof. Levent Eren and Prof. Murat Aşkar for their tremendous assistance and valuable suggestions throughout my graduate studies.

My deepest appreciation goes to the Scientific and Technological Council of Turkey (TUBITAK) for the award-winning support "2210-A National Scholarship Program for MSc Students".

Finally, I would like to thank my parents and brother for their love, support, and patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

CHAPTER 1: INTRODUCTION

Electrical machines are found in various industrial and commercial settings. The global electric motor market size was USD 106.45 billion in 2020. The market is estimated to grow from USD 113.14 billion in 2021 to USD 181.89 billion by 2028 at a compound annual growth rate (CAGR) of 7.0% during 2021-2028 (Industry Report, 2021). The increase in worldwide power consumption and the increasing use of electrical machinery are the important aspects driving growth in the global electric motor market. The market size has also been increasing with the development of renewables sector and green transportation.

Induction motors are the most common and frequently encountered machines in industrial, commercial, and residential settings. They are the essential components of industry due to their broad use in wide range of applications such as cooling, heating, and pumping. They are advantageous thanks to their low cost, simple and sturdy design, easy maintenance, and high power-to-weight ratio. Thanks to these factors, ac induction machines are dominant in industry, representing more than 90 percent of installed motor capacity (Ferreira and De Almeida, 2012).

Electric motor predictive maintenance aims to predict and correct motor faults before they become expensive and affect productivity. Rolling element bearings (REBs) are machine elements that are usually found in rotating equipment, and they are designed to support a load while minimizing friction. If they break suddenly, a catastrophic failure may occur resulting in associated high repair and replacement costs.

REBs are mostly encountered in rotating machinery, but the bearings are not free from failure. Records indicate that faulty bearings contribute to around forty percent of induction motor failures (Ergin, Uzuntas and Gulmezoglu, 2012). They are the most significant reasons for machine breakdowns; thus, the lifetime and condition of a rolling element bearing are of interest to customers to sustain continuous operation. Early diagnosis of bearing defects allows bearing replacement rather than

complete electric motor replacement and it enhances safety, efficiency, reliability, and availability, resulting in reduced unexpected costs. However, although the replacement of faulty bearings is the most cost-effective solution, it is the most difficult one to detect. Accelerometers, thermocouples, microphones, piezo-velocity sensors, current sensors and many more devices could be used to detect bearing faults and researchers have been using these sensors for vibration monitoring, temperature monitoring, and acoustic emission monitoring of the rotating machines (Correa and Guzman, 2020). Using these condition monitoring techniques, many studies are also ongoing to build intelligent bearing fault detection and diagnosis (BFDD) systems.

Signal-based, model-based, and knowledge-based approaches could be implemented for BFDD systems. Model-based methods require expert knowledge on the system dynamics, and it mainly uses predictions generated using mathematical models describing the healthy response of a system. A fault diagnosis algorithm then makes a decision on the status of the system using the model's predicted output and the measured output. Fault diagnosis using model-based approach is straightforward if the model has no complex mapping with physical parameters, but the accuracy mostly increases with complexity of model representation. The vast majority of motor fault detection and diagnosis systems typically process raw motor vibration or current data. Moreover, in signal-based approaches, no explicit motor model is used in any of the signal processing stages. Various time, time-frequency and frequency domain signal processing techniques have been employed in signal-based fault diagnosis systems. These systems are prone to unknown or unbalanced conditions. In signal-based systems, fault detection accuracy is usually increased with advanced signal processing tools, but they often result in increased computational complexity. Unlike model-based methods, which rely on expert knowledge, the knowledge-based approach is data-oriented and makes no assumptions about the physical model of the system.

The knowledge-based approaches are investigated in two main groups: quantitative approaches based on machine learning and qualitative methods based on symbolic intelligence. Quantitative approaches could further be grouped into two main categories: unsupervised learning systems such as principal component analysis (PCA)

and K-means, and supervised learning systems such as support vector machines (SVMs), artificial neural networks (ANNs) and convolutional neural networks (CNNs). The diagnosis accuracy of knowledge-based approaches is strongly influenced by the quality of training data.

Bearing failures often occur at specific fault frequencies in frequency spectrum and signal-based BFDD systems employ these frequencies to estimate the likelihood of a bearing problem. Furthermore, conventional machine learning models include a feature extraction mechanism to get a meaningful representation of the data, and it is followed by a classification stage that determines the condition of bearings from the extracted features. Even though satisfactory results of fault diagnosis accuracies were stated in the prior studies, various features and classifiers were used for this purpose. However, selecting the right feature is often quite troublesome, and manually selected or handcrafted features often result in information loss. Furthermore, as feature extraction often brings computational complexity to the system, its use in real-time condition monitoring applications may not be appropriate. Consequently, to avoid these limitations and achieve better performance at noisy environments, researchers have implemented several deep learning (DL) based methods that uses raw motor current or vibration data for early diagnosis of bearing defects.

Ince et al. (2016) first proposed a unique method that utilizes 1D CNNs for BFDD. This work demonstrated CNNs' ability to learn to extract discriminative features from the training data using a set of 1D filter kernels. In this model, following convolutional layers, a multilayer perceptron (MLP) was utilized to perform the classification task. The proposed method only performs hundreds of 1D convolutions to construct the output decision vector, making it suitable for real-time BFDD systems. As an extension of this work, 1D Self-organized Operational Neural Networks with generative neurons (1D Self-ONNs) was again proposed by (Ince et al., 2021). With this study, the superiority of 1D Self-ONNs over 1D CNNs was shown for the bearing fault severity level classification problem. The details of these studies along with the other deep learning-based BFDD approaches are given in Chapter 3.

In this thesis, a newly emerging paradigm, tiny machine learning (TinyML), is utilized for BFDD problem. TinyML aims to shrink and run deep neural networks on ultra-low power microcontrollers. First, shallow 1D CNN and Self-ONN models were trained and tested on Case Western Reserve University (CWRU) and University of Ottawa motor vibration data. Then, from two different single-phase induction motors, real motor vibration data was collected for healthy, outer-race, inner-race, and ball bearing fault conditions. Using this newly collected dataset, a shallow 1D CNN model was trained, quantized, and deployed to Arm Cortex-M4 based ultra-low-power STM32L4 series microcontroller. For this purpose, the two most popular embedded AI frameworks STM32Cube.AI and TensorFlow Lite for Microcontroller (TFLM) were used, and they were evaluated regarding test accuracy, average inference duration and memory footprint for this application. Finally, a large number of experiments were performed by inputting 3-axis real motor acceleration data to the deployed 1D CNN model to show that it is computationally feasible to detect motor bearing faults in real-time using low-power microcontrollers.

The rest of this thesis is organized as follows. Chapter 2 offers an introduction to rolling element bearing failure, fault stages and corresponding fault frequencies. State-of-the-art deep learning methods for bearing fault diagnosis are discussed in Chapter 3. Chapter 4 describes the CWRU and University of Ottawa bearing datasets used to train the 1D CNN and Self-ONN models and provides training and test results. Quantization and deployment of neural networks on microcontrollers are further studied in Chapter 5. Chapter 6 proceeds with experimental setup and on-device performance results. Finally, a conclusion of this study with a short summary and possible future work is given in Chapter 7.

CHAPTER 2: ROLLING ELEMENT BEARING FAILURE

2.1 Failure Stages

Rolling element bearings (REBs) are mechanical components present in almost all rotating machinery and they are undoubtedly one of the most abundant elements in industry. REBs can be found in everything from electric motors to conveyor systems and gearboxes. If a shaft needs to rotate, it is most likely supported by a REB. Their primary goal is to reduce rotational friction while also supporting radial or axial loads. REBs are composed of an inner ring and race, an outer ring and race, a set of rolling elements and the cage, as shown in Figure 1. In many applications, the outer ring is stationary, and the inner ring carries the rotating shaft. However, in some cases, while the outer ring rotates, the inner ring stays stationary. The cage is used to keep the rolling elements apart and evenly spaced. The shape of the rollers between the two rings determines the load a certain bearing can withstand, and this shape also affects the lubrication requirements. The most widely used type is the ball bearing and it is used for moderate loads.

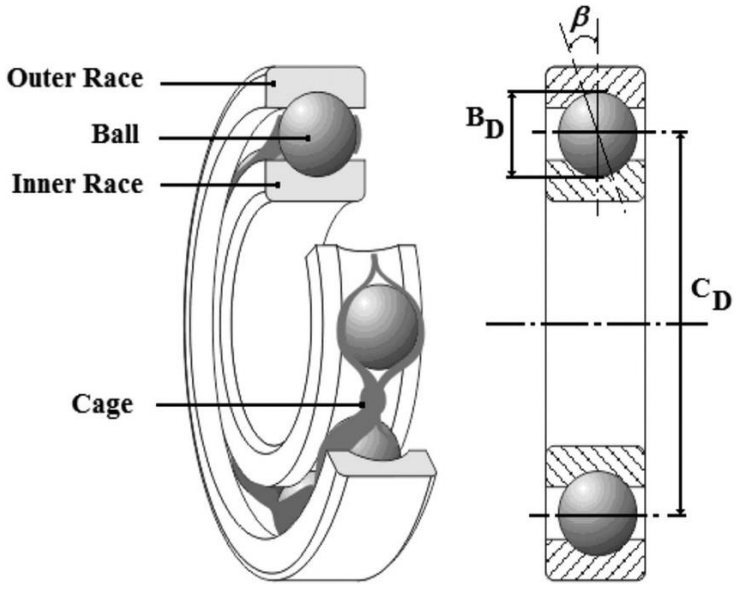

Figure 1. Rolling element bearing components.

In general, even if the installation and maintenance of REBs are performed properly, metal fatigue failure is still inevitable. At the same time, since bearings often work under harsh conditions, other sources of failures such as loss of lubrication, corrosion, contamination, and overheating may also exist (Bloch and Geitner, 1999).

Based on the vibration frequencies generated by rolling elements hitting the defects in the outer and/or inner races, bearing faults can be grouped into four main failure stages. These four stages were defined by Technical Associates of Charlotte, P.C. (Harris, 2001) and each stage is discussed below.

At the first stage of bearing failure, small pits start to appear in the inner and/or outer bearing race and impacts of rolling elements to these defects result in vibration activity at ultrasonic frequency range from around 20 kHz to 350 kHz. In stage 1, bearings should still be operating normally, and they do not need to be replaced. Even though there is no need for bearing replacement, stage 1 often gives the indication of lack of lubrication between the rolling elements and races, and if the bearing defects progresses in time, the amplitude peaks begin to appear at lower frequencies in the spectrum.

In stage 2, when the bearing defects become larger, they start to ring at the natural frequency of the bearing components. At this stage, vibration activity can be observed in the frequency range from 500 Hz to 2 kHz. These resonance frequencies may occur due to bearing support structures or due to the components of the bearings themselves such as rolling elements and races. Progressive bearing wear usually generates sideband frequencies above and below the component or casing natural frequencies.

In stage 3, if the bearing is removed, the defects and wear patterns can be clearly observed in the raceways. Bearings reaching this stage should be replaced in a short time for both critical and non-critical machinery. At this stage, characteristic frequencies may now be observed from the frequency spectrum. As wear progresses in this stage, well-formed sidebands accompany the fault frequencies and harmonics

(Harris, 2001). Furthermore, sidebands around the bearing component natural frequencies and the high frequency content evident in stages 1 and 2 continue to grow and become more evident at stage 3.

Stage 4 takes place at the end of a bearing's lifetime. In this stage, as the bearing defects progresses, rotor vibration increases, and rotor-related frequencies become dominant in the spectrum. A substantial decrease in the amplitude of the bearing component natural frequencies is also observed. At the same time, random high frequency vibration occurs, causing an increased noise floor (Harris, 2001). Bearings reaching this stage should be replaced immediately, otherwise with damage to other machine components, a catastrophic failure might happen.

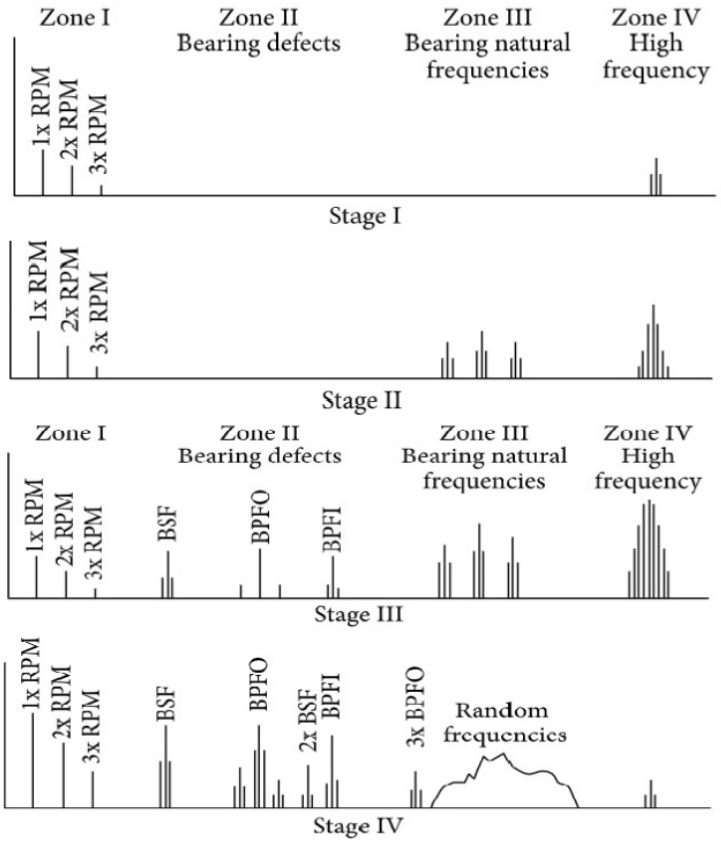

Figure 2. Bearing fault stages (Source: Eren, 2017).

In Figure 2, the frequency spectrum for all stages is shown. BPFI, BPFO and BSF represent ball pass frequency inner race, ball pass frequency outer race and ball spin frequency, respectively.

2.2. Bearing Fault Frequencies

The bearing fault characteristic frequencies are calculated using the bearing geometry and shaft speed. Figure 3 depicts the geometry of a commonly used ball bearing.

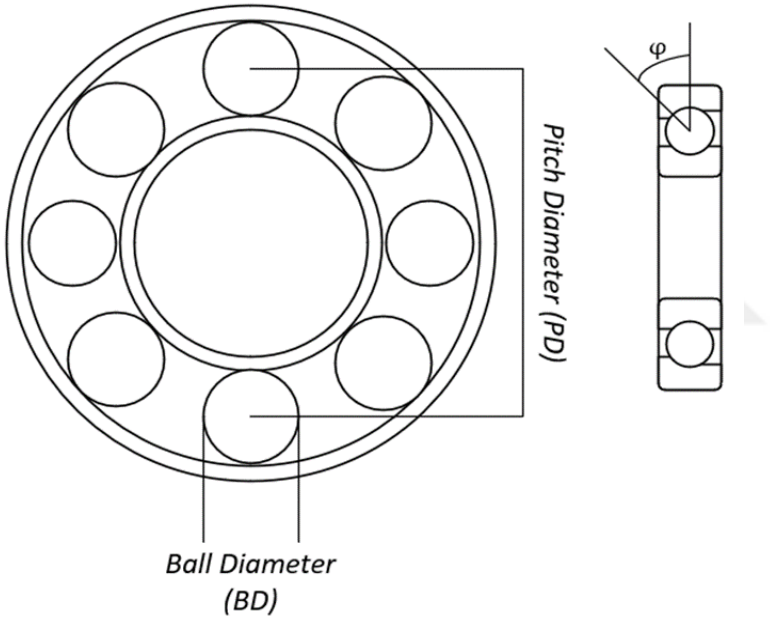

Figure 3. The geometry of a ball bearing (Source: Eren, 2017).

When one of the bearing components has a defect, a peak appears at a specific characteristic frequency. These vibration frequencies representing various fault locations are called Ball Pass Frequency Outer (BPFO) or outer-race fault frequency, Ball Pass Frequency Inner (BPFI) or inner-race fault frequency, Ball Spin Frequency (BSF) or ball fault frequency, and Fundamental Train Frequency (FTF) or cage fault frequency. The number of balls that pass through a specific location of the outer race during a full rotation of the shaft corresponds to BPF0, which is given by the equation:

$$BPFO = \frac{N_B . \mathrm{f}_r}{2}\left(1 - \frac{BD}{PD}\cos\varphi\right)$$

Equation 1. Ball Pass Frequency Outer (BPFO).

In this equation, f_r represents rotor speed in revolutions per second, N_B is the number of rolling elements (i.e., balls), and φ is the contact angle of the load from the radial plane (zero for ball bearings). On the other hand, Ball Pass Frequency Inner (BPFI) corresponds to the number of balls that pass through a given point of the inner race each time the shaft makes a complete turn, and BPFI can be calculated as:

$$BPFI = \frac{N_B . \mathrm{f}_r}{2}\left(1 + \frac{BD}{PD}\cos\varphi\right)$$

Equation 2. Ball Pass Frequency Inner (BPFI).

Ball Spin Frequency (BSF) is the number of turns that a bearing ball makes during a full rotation of the shaft, and it is expressed as:

$$BSF = \frac{\mathrm{PD}}{2\mathrm{BD}}\, f_r\left(1 - \left(\frac{BD}{PD}\cos\varphi\right)^2\right)$$

Equation 3. Ball Spin Frequency (BSF).

Finally, Fundamental Train Frequency (FTF) is the number of turns a bearing cage makes during a full rotation of the shaft, and it is given by the following equation:

$$FTF = \frac{1}{2} f_r\left(1 - \frac{BD}{PD}\cos\varphi\right)$$

Equation 4. Fundamental Train Frequency (FTF).

2.3. Sensor Selection

Accelerometer sensors are perhaps the only sensors that can monitor the condition of a REB for all fault stages. They are usually recommended as they have the following advantages. First, by integration of acceleration signal, velocity information can be obtained for the evaluation of fault stages 3 and 4. For the earliest indication of a bearing fault, it is recommended to use an accelerometer sensor that has a high frequency range. A lubrication problem in the early stages, i.e., stage 1 and 2, often results in a bearing failure in the long run, and a wideband accelerometer may provide early signs of this problem avoiding further deterioration and maintenance costs. Velocity information can also be used to detect and correct problems such as misalignment and unbalance that may cause a bearing failure. Although the installation of accelerometers is relatively easy, their performance (e.g., frequency response) may degrade depending on the mounting location and installation techniques.

Current sensors may also be utilized to diagnose bearing faults. However, they are often useful for stage 3 and 4 bearing defects, because the rotor and stator relationship is highly affected by the progressive bearing failure in these stages. Hence, current sensors are less sensitive than accelerometers for early stages of bearing failures.

When a bearing protection is needed, temperature measurement might be considered. Resistive Temperature Devices (RTDs) and thermocouples are often used for bearing temperature monitoring. Temperature increase on the bearing could be detected when the bearing fault has reached to stage 4. However, different aspects such as the ambient temperature and air flow can affect the temperature readings, thus they should be considered when assessing the condition of the bearings. Temperature sensors are usually cheaper and physically smaller than vibration transducers, and they require much less power. Rolling element bearing P-F curve indicating the earliest fault detection point for different sensors is illustrated in Figure 4.

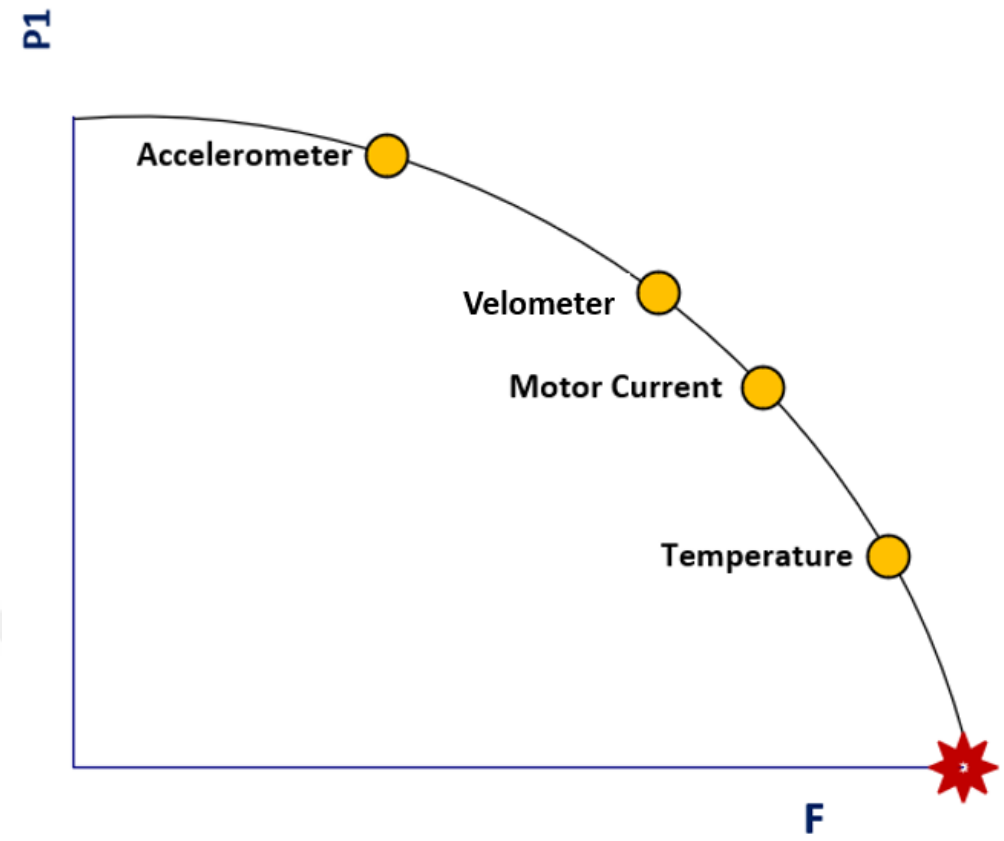

Figure 4. REB P-F curve indicating the earliest fault detection point for different sensors.

CHAPTER 3: DEEP LEARNING BASED APPROACHES FOR BEARING FAULT DIAGNOSIS

3.1. Autoencoders

As a specific type of feedforward neural networks, an autoencoder is commonly utilized as unsupervised learning mechanism that aims to transform its inputs to outputs with minimum distortion. Autoencoders can be encountered in numerous applications such as dimensionality reduction, feature extraction, image compression, denoising and generation. They were first proposed by Hinton and the PDP group (Rumelhart, Hinton and Williams, 2013), and employed to solve the problem of backpropagation without ground truth labels by using the input data as labels. In recent years, researchers have been using auto-encoders as an unsupervised feature extraction method and as a greedy layer-wise neural network pre-training method to avoid vanishing gradients in layers close to the input and allow very deep neural networks achieve improved performance.

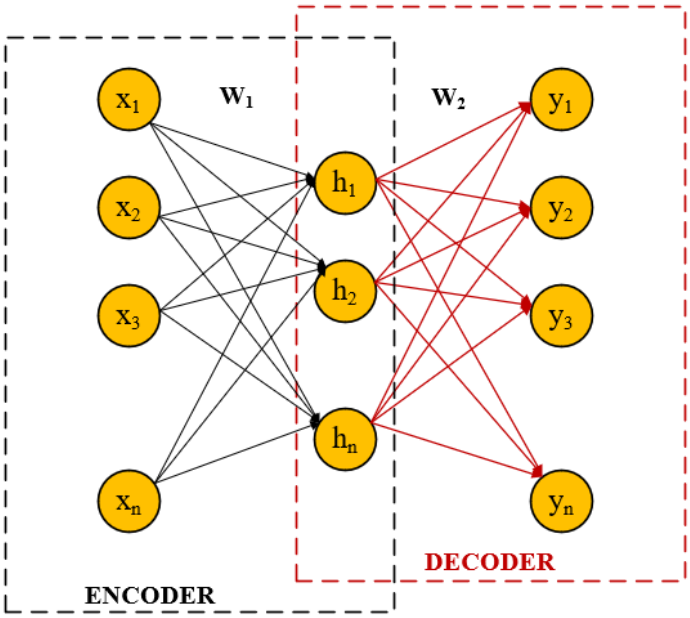

Figure 5. Autoencoder architecture.

As illustrated in Figure 5, the autoencoder topology contains two main parts: encoder and decoder. They are both fully-connected feedforward neural networks. The encoder network can be represented by an encoding function p_θ. From the training dataset $\{\boldsymbol{x}^m\}_{m=1}^M$, each signal $\boldsymbol{x}^m$ is encoded to get a code representation as:

$$\boldsymbol{h}^m = p_\theta(\boldsymbol{x}^m)$$

Equation 5. Autoencoder encoding function.

where $\boldsymbol{h}^m$ is the encoded vector in the hidden layer obtained using $\boldsymbol{x}^m$.

Then, the decoding function $q_{\theta'}$ maps the code $\boldsymbol{h}^m$ from lower dimensional space back into higher dimensional space to produce reconstructed output vector $\tilde{\boldsymbol{x}}^m$ as given in Equation 6.

$$\tilde{\boldsymbol{x}}^m = q_{\theta'}(\boldsymbol{h}^m)$$

Equation 6. Autoencoder decoding function.

The parameters are learned to reconstruct the input as original output with a minimum loss $L(\boldsymbol{x}, \tilde{\boldsymbol{x}})$ (usually mean-squared-error) for the M training examples. Affine mappings are often used for both the encoder and decoder parts, and they are followed by nonlinear activation functions as given in equations 7 and 8.

$$p_\theta(\boldsymbol{x}) = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

Equation 7. Autoencoder encoder mapping.

$$q_{\theta'}(\boldsymbol{x}) = g(\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x} + \boldsymbol{c})$$

Equation 8. Autoencoder decoder mapping.

where f and g are the encoder and decoder activation functions, respectively. Therefore, weight matrices $\boldsymbol{W}$ and $\boldsymbol{W^T}$, and the bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ are the learned parameters for an autoencoder architecture.

In Figure 5, an autoencoder with one hidden layer is given. This hidden layer defines a code representing the corresponding input, and this code is further fed into the decoder. The size of the hidden layer, i.e., code size, is a hyperparameter which we can decide before training the autoencoder. For the training, the network usually takes the mean-squared-error of the output and the original input to generate an output similar to the input. Once the network is trained, the encoder is kept but the decoder part is removed. Therefore, the encoder's output contains a sparse (feature) representation of the input which may be used in subsequent classifiers.

Many studies utilizing autoencoders for bearing diagnosis have been published in the literature. One of the first studies was stacked autoencoders by Jia et al. (2016). In this work, the input to the stacked autoencoder model was the frequency spectra of the raw vibration signal. This stacked autoencoder based DNN was pre-trained layer by layer sequentially in an unsupervised way, and the output layer of the network was used for classification. In this study, the sub-datasets A (1 hp), B (2 hp), C (3 hp) and D (1-3 hp) of CWRU bearing data was used to assess the performance of this model, and the number of classes is 10, which includes the severity and location of the bearing fault. This method achieved a high testing accuracy above 99% in all sub-datasets.

Lu et al. (2015) also used a DNN architecture formed by stacked autoencoders. First, they trained an autoencoder with the input set, and obtained the corresponding feature vector. Then, the output layer of the first autoencoder was removed and the feature vector was used as input set for the next autoencoder. By iteratively executing these steps, a DNN structure could be formed. This work used the CWRU bearing data with 3 bearing fault locations (IR, OR and BF) and 2 fault sizes (0.007 and 0.014 inches) under no load condition (0 hp), thus the number of classes was 6. The input vibration data was preprocessed in this work. First, the data was segmented into 600 time-domain samples with an overlap of 80%. Then, by taking the Fast Fourier

Transform (FFT) of the input signal, its amplitude of spectrum was obtained. Since the amplitude of frequency was too small, they multiplied the coefficients by 10. Once the feature vectors of the DNN was obtained, PCA method was implemented to reduce the dimension of the data for visualization.

Some other studies that utilize autoencoders for bearing fault diagnosis on CWRU dataset were conducted by Lu et al. (2017), (Guo, Chen and Shen, 2016), and (Zhang et al., 2019), and the researches by (Shao et al., 2017) and (Wang et al., 2018) are some examples that uses autoencoders on different datasets.

3.2. Generative Adversarial Networks

Generative Adversarial Networks (GANs) were first introduced by (Goodfellow et al., 2014), and they have gained tremendous interest in deep learning. In an adversarial learning process, a generative model G captures the data distribution, whereas a discriminative model D is used to estimate the probability that a sample came from the training data rather than G. GANs are composed of two neural networks, i.e., discriminator D and generator G, as shown in Figure 6. With the value function given in Equation 9, this topology is equivalent to a two-player minimax game.

$$Min_G Max_D V(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

Equation 9. Minimax value function for GANs.

In this equation, $D(x)$ is the discriminator's probability estimation of real data instance x being real, $G(z)$ is the generator's output when noise z is given as input, $D(G(z))$ is the discriminator's estimate of the probability that a fake instance is real, and E means the expected value. The equation comes from the cross-entropy between the generated and real distributions. The generator does not have a direct effect on the $\log D(x)$ term in the value function, thus minimizing the loss is the same as minimizing $\log(1 - D(G(z)))$ for the generator.

In a GAN architecture, the generator takes in random noise and gradually learns to output fake data by leveraging feedback from the discriminator throughout this two-player minimax game. Generator's aim is to make the discriminator classify its output as real. The discriminator, on the other hand, is a simple classifier that aims to discriminate real data from the data created by the generator. Depending on the type of data its classifying, any network architecture can be used in the discriminator part.

The discriminator's training data is made up of real data instances and fake data instances created by the generator as shown in Figure 6. The discriminator uses real data instances as positive examples, and fake data instances as negative examples during training. Consequently, in a GAN training process, the generator tries to reach real data distribution, and the training eventually reaches Nash equilibrium, where the discriminator has a 50% accuracy.

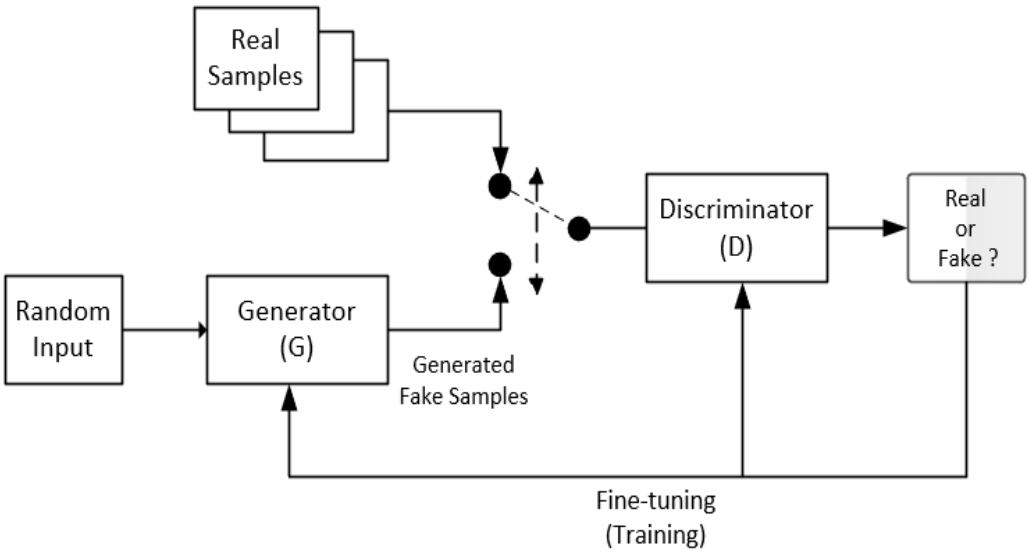

Figure 6. Architecture of GANs.

Many studies in the literature have used GANs to diagnose bearing faults. They are especially useful when we want to train a model on a source dataset and then test it on a target data which has different distribution. Domain adaptation is required to reduce dataset bias and improve generalization capability of a deep learning model.

(Cheng et al., 2020) proposed a Wasserstein Distance-based Deep Transfer Learning (WD-DTL) network. With an adversarial training process, WD-DTL seeks domain-invariant features by a CNN and minimize the distribution between source and target domain. They implemented Wasserstein-1 distance between two different feature distributions through adversarial training by employing a domain alignment critic. In this study, the authors used 4 different kinds of bearing health condition (healthy, outer-race, inner-race, and ball fault) from CWRU dataset. As a pre-processing step, power spectrum of input signal was computed, and the left side was clipped. Three different scenarios were considered. In the first case, the authors tried to transfer knowledge between different motor loading conditions and obtained 95.75% average accuracy across all load domains. In the second scenario, they attempted unsupervised domain adaptation between two sensor locations (fan-end and drive-end) and achieved 64.20% average accuracy. In the last scenario, the same setting was used as in the previous one, but 0.5% of labeled data from target domain was introduced to the source data to enhance the performance (average accuracy of 64.92%).

(Zhou et al., 2020) utilized DNN's feature extraction capabilities and GAN's data generating capabilities to address dataset imbalances. The bearing fault diagnosis approach in this study is built upon global optimization GAN. The generator was designed such that it generates features for unbalanced class samples using some labeled fault samples by an autoencoder. The training process of the generator was guided by fault feature and fault diagnosis error, and the discriminator was designed to filter unqualified generated samples from the qualified ones for more accurate fault diagnosis. For data imbalance ratio of 10:1, the diagnosis accuracy was 94.58%, 96.85%, and 93.28% for inner-race, ball and outer-race fault, respectively.

Some other studies utilizing GANs for bearing fault diagnosis on CWRU dataset were conducted by (Jiang et al., 2019), (Zhao, Liu and Meng, 2019), and (Mao et al., 2019), and the studies by (Li et al., 2019) and (Gao et al., 2019) are some examples that use GANs on different dataset.

3.3. Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a kind of ANN that allows previous outputs to be processed as inputs while having hidden states. RNNs contain a built-in feedback loop, which allows them to remember current and past information when arriving at a decision. RNNs are often preferred when data is sequential, and the next data depends on the previous data point. Therefore, they are often used in text, speech recognition and natural language processing applications. Typical RNN architecture is shown in Figure 7. For each timestep t, the activation $a^{\langle t \rangle}$ and the output $y^{\langle t \rangle}$ are expressed as given in Equation 10 and Equation 11.

$$a^{\langle t \rangle} = g_1\left(W_{aa}a^{\langle t-1 \rangle} + W_{ax}x^{\langle t \rangle} + b_a\right)$$

Equation 10. The activation for an RNN.

$$y^{\langle t \rangle} = g_2\left(W_{ya}a^{\langle t \rangle} + b_y\right)$$

Equation 11. The output for an RNN.

where g_1, g_2 represent the activation functions and $W_{aa}, W_{ax}, W_{ya}, b_a, b_y$ are coefficients shared temporally.

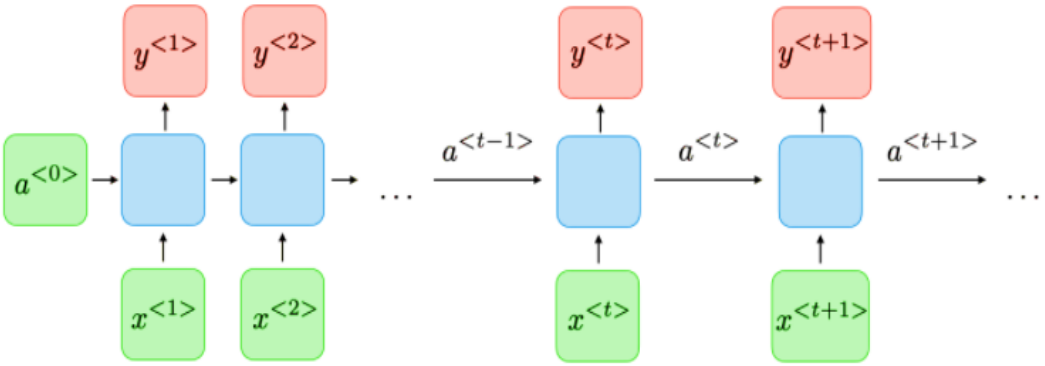

Figure 7. Architecture of a typical RNN.

In an RNN, the exploding or vanishing gradients problem is often encountered. Due to the multiplicative gradient, which might be exponentially increasing or decreasing with the number of layers, it is difficult to capture long-term dependencies. For that reason, even though they were first introduced in 1980s, they had limited applications. In order to cope with the exploding gradient problem, gradient clipping is often performed by limiting the maximum value of the gradient. On the other hand, to solve the vanishing gradient problem encountered by traditional RNNs, Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) were introduced. As a generalization of GRU, LSTM architecture is extended by adding specific gates especially forget gate. An LSTM can then forget and remember patterns selectively for a long duration of time. In Figure 8, the internal structure of an LSTM cell is shown. In this figure, Γ_u, Γ_r, Γ_f, and Γ_o are used for the update gate, relevance gate, forget gate and output gate, respectively. Each gate could be described by the Equation 12.

$$\Gamma = \sigma\left(Wx^{\langle t\rangle} + Ua^{\langle t-1\rangle} + b\right)$$

Equation 12. The general gate equation in an LSTM.

In the Equation 12, σ is the sigmoid function, and W, U and b are the coefficients specific to the gate. Then, the characterizing equations of an LSTM architecture are summarized in the following equations as:

$$\tilde{c}^{\langle t \rangle} = \tanh\left(W_c\left[\Gamma_r * a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_c\right)$$

Equation 13. Cell update equation in an LSTM.

$$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + \Gamma_f * c^{\langle t-1 \rangle}$$

Equation 14. Cell state equation in an LSTM.

$$a^{\langle t \rangle} = \Gamma_o * c^{\langle t \rangle}$$

Equation 15. Output equation in an LSTM.

In these equations, the element-wise multiplication between two vectors is denoted by $*$ sign.

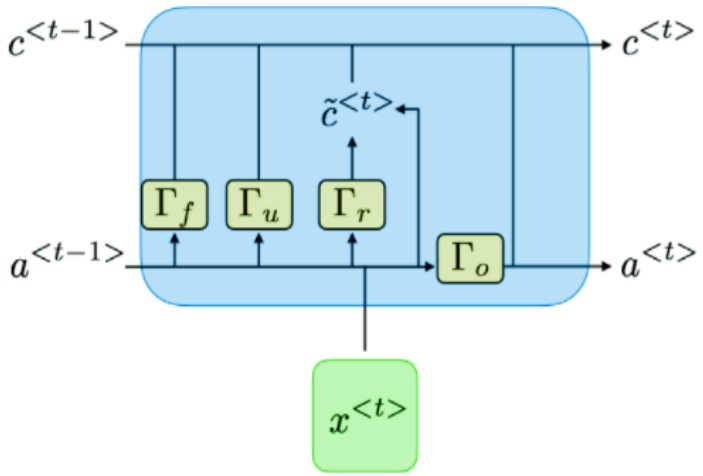

Figure 8. Internal structure of an LSTM cell.

Pan et al. (2018) proposed a BFDD method based on the combination of 1D CNN and LSTM architectures. The input to the model was the raw data. The architecture was formed by a 1D convolutional, and a pooling layer followed by an LSTM layer, and a SoftMax at the output of this classifier. The model was evaluated for a single loading condition, and the average accuracy rate was over 99%.

Liu et al. (2018) proposed an RNN in the form of an autoencoder for BFDD. Using a Gated Recurrent Unit (GRU)-based denoising autoencoder, this approach attempts to predict multiple vibration values of the bearings for the upcoming period from the previous one. This GRU-based denoising autoencoders were trained for each fault class. Then, for an input, the reconstruction error between the output data from the network and the next period data were utilized to spot an anomaly and classify bearing defects. The diagnosis accuracy of this method was higher than 96% even for 1 dB SNR. (Jiang et al., 2018) and (Zhuang et al., 2019) also use RNN and LSTM for BFDD problem on CWRU dataset.

3.4. Convolutional Neural Networks

Inspired by the mammalian visual system, a Convolutional Neural Network (CNN) that can be trained using backpropagation was first proposed by (LeCun et al., 1989) as a space invariant artificial neural network built upon the shared-weight architecture of the convolution kernels. CNNs gained immense popularity when they exhibited superior performance compared to other models at ILSVRC (ImageNet Large Scale Visual Recognition Challenge). They are feedforward neural networks and have the following four layers: input, convolutional, pooling and fully-connected layers. An example input may be an image for a 2D CNN or raw vibration data for a 1D CNN. Convolutional layers of a CNN are used to extract discriminative features using a collection of convolution kernels. Two properties “weight sharing” and “limited connectivity” of convolutional layers separate CNNs from the conventional MLPs. After sliding the input features through the convolutional kernels, pooling layers could then be employed to decrease the size of feature maps, resulting in reduced number of parameters and computation in the network. Final layers of CNNs are fully-

connected MLPs, and they are used for the purpose of classifying extracted features. In the following section, the most popular 2D CNNs and the recent compact and adaptive 1D CNN architecture will be discussed in detail. However, this section will mainly focus on adaptive 1D CNNs since they provide various advantages and superiorities over 2D CNNs, especially for 1D data.

3.4.1. 2D Convolutional Neural Networks

In contrast to traditional MLPs which have scalar weights, input, and output, in 2D CNNs, each neuron has 2D planes for weights called kernels, as well as 2D input and output called feature maps. A sample 2D CNN configuration is illustrated in Figure 9. A 28 × 28-pixel single channel image is inputted to this network and the image is classified into two categories at the output. The network is composed of 2 convolutional and pooling layers with 4 and 8 neurons, respectively. In this sample 2D CNN configuration, the first convolutional layer has 4 filters with the kernel size of (Kx=8, Ky=8), while 8 filters having a kernel size of (Kx=4, Ky=4) are used in the second convolution layer. In the convolutional layers, with convolution stride size of 1 and no padding, the width and height of the input feature map will be reduced by (Kx-1, Ky-1) pixels, respectively. The feature map size will further be reduced by the amount of subsampling factors which are chosen as (Sx=3, Sy=3) for the first pooling layer and (Sx=4, Sy=4) for the second pooling layer. A dense layer with four neurons follows the last pooling layer. The output layer generates classification output, and the number of neurons is equal to the number of classes.

Forward propagation through this sample 2D CNN occurs in the following order. Firstly, A 28 × 28-pixel single channel image is inputted to this network. The input feature map of each neuron of the first convolutional layer is then produced by performing a linear convolution operation between this image and the associated filter.

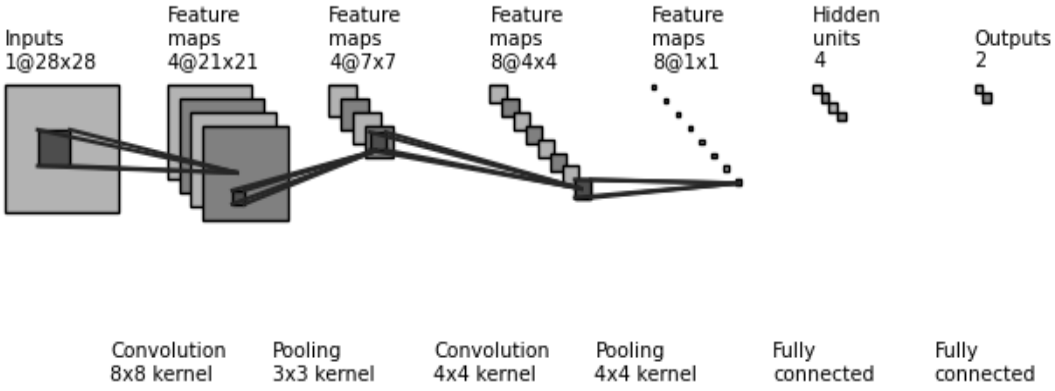

Figure 9. An example of 2D CNN configuration.

Once the convolution operation is performed, the input feature map of each convolutional neuron is passed through the activation function to produce the output feature map. This output feature map is now decimated by 3 × 3 kernel creating 7 × 7 feature maps as a result of the first pooling layer. The same steps are repeated for the second convolution and pooling layers as the first one, and the second pooling layer's scalar outputs are finally inputted and forward propagated through the fully-connected and output layers to generate the final output that shows the classification result of the input image.

The backpropagation (BP) algorithm is generally used in a CNN training process. The gradient magnitude of each network parameter (e.g., the weights of the convolutional and dense layers) is computed in each iteration of BP. Then, again in each iteration, these gradient magnitudes are used to update network parameters. Various gradient-descent optimization algorithms such as Adam (Kingma, 2015), Stochastic Gradient Descent (SGD) and SGD with momentum (Qian, 1999) are implemented in BP. (Kiranyaz, Ince and Gabbouj, 2016) provides a full explanation of BP algorithm for 2D CNNs.

(Guo, Chen and Shen, 2016) developed a novel hierarchical learning rate adaptive deep CNN (ADCNN) to diagnose bearing faults and predict their severity. In this paper, an adaptive learning rate and a momentum component was added to CNN

architecture to avoid training failure. The first layer of this ADCNN model was based on the classical LeNet5 model. The number of classes is four and includes the bearing health conditions healthy, outer-race, inner-race, and ball fault from CWRU dataset. The mean validation accuracy was 97.9% for 10-fold cross validation. The fault size was evaluated in the second layer of ADCNN. In each fault class, the bearing fault severity classification accuracy was above 99%.

Xia et al. (2018) suggested a CNN-based bearing fault diagnosis approach that incorporates sensor fusion. 1D raw data from multiple sensors were layered row by row to construct a 2D matrix at the input of the proposed 2D CNN. The model was evaluated on CWRU bearing data. The 2D matrix at the input of this model was formed using the vibration signals from fand-end, drive-end, and base of the motor. These waveforms are stacked to form a 2D matrix. The average accuracy of 99.41% was achieved with multiple sensors, but the accuracy dropped to 98.35% when a single accelerometer was used.

There are many studies utilizing CNNs for BFDD on the CWRU bearing dataset, and some of them are (Zhang et al., 2020), (Hoang and Kang, 2019), and (Zhang, Peng and Li, 2017).

3.4.2. Adaptive 1D Convolutional Neural Networks

2D CNNs are very useful for two-dimensional data, but a modified form of 2D CNNs called 1D Convolutional Neural Networks (1D CNNs) can rather be used for one-dimensional data. In the literature, 1D CNNs have been shown to outperform 2D CNNs in some applications that have limited labeled data with high variations (Kiranyaz et al., 2021). Training and inference duration can be significantly reduced using 1D CNNs with shallow architectures due to their low computational requirement. Therefore, they are ideal for real-time and low-cost applications.

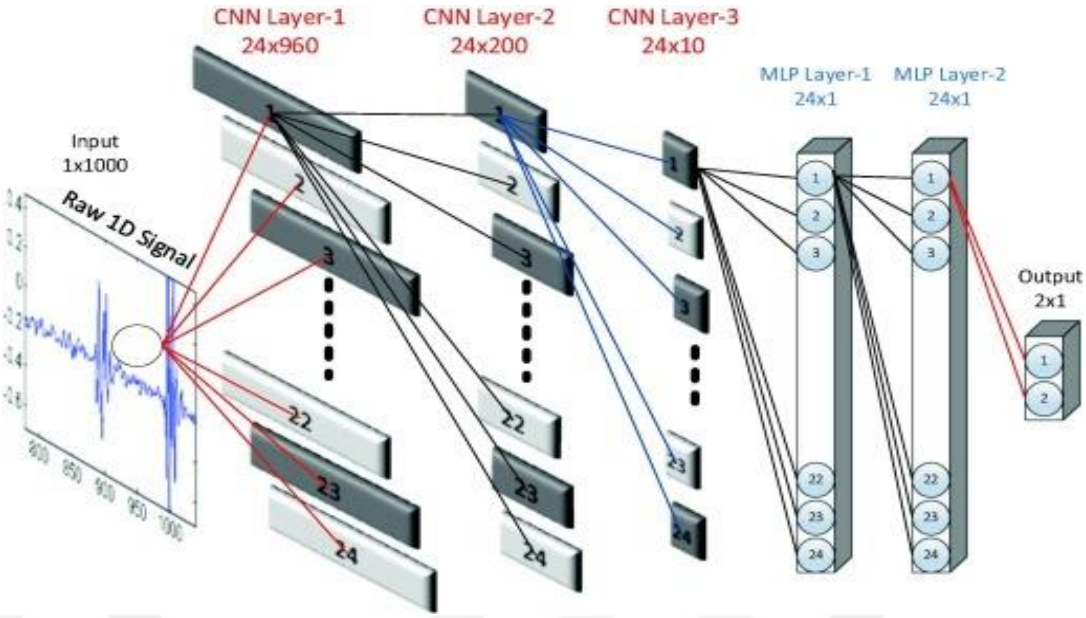

Figure 10. The illustration of a sample 1D CNN configuration (Source: Kiranyaz et al., 2021).

As demonstrated in Figure 10, 1D CNNs are made up of two unique layer types: 1D convolutional layers and dense layers. 1D convolutions, activation function and sub-sampling (pooling) operations take place in the CNN layers. The dense layers are also known as MLP layers since they are the same as the layers of a simple Multi-layer Perceptron (MLP). Moreover, the input layer is just a passive layer that accepts raw one-dimensional data, and the output layer has the same number of neurons as the number of classes.

The convolutional layers of a 1D CNN learn to extract discriminative features by processing raw 1D input, and these features are then used in the dense layers for classification. As a result, both feature extraction and classification tasks are combined in a 1D CNN to optimize classification performance. Some hyperparameters to tune are the number of hidden CNN and MLP layers and neurons, filter (kernel) size and subsampling factor in each convolutional layer, and the choice of pooling and activation functions. For the sample 1D CNN configuration given in Figure 10, there are three hidden convolutional and two hidden dense layers, subsampling factor is 4, and the filter size is 41 in all hidden convolutional layers.

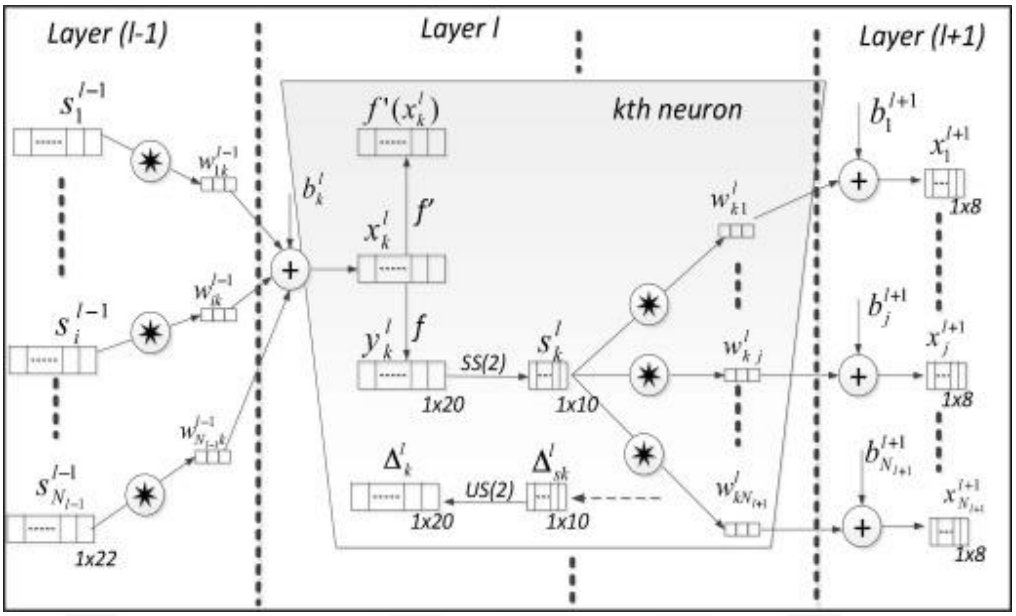

Figure 11. Three successive convolutional layers of a 1D CNN (Source: Kiranyaz, Ince and Gabbouj, 2016).

In place of the 2D matrices used in 2D CNNs for kernels and feature maps, 1D arrays are formed at each neuron's input and output in the successive CNN layers as a consequence of 1D convolution and subsampling. Hence, 2D matrix operations such as 2D convolution (*conv2D*) and lateral rotation (*rot180*) are now replaced by their 1D counterparts, *conv1D* and *reverse,* in the forward and back-propagation phases of 1D CNN architecture (Kiranyaz et al., 2021). Therefore, the only operation with a significant computational cost is a sequence of 1D convolutions, i.e., linear weighted sums of two 1D arrays, and these operations during the forward and back-propagation could effectively be executed in parallel (Kiranyaz et al., 2021). Figure 11 depicts three successive convolutional layers of a 1D CNN. A sequence of 1D convolutions with 1D filter kernels of size 3 are first performed at the k^{th} neuron in the hidden CNN layer *l*. Then, the sum is passed through the activation function *f,* whose output is subsampled by the subsampling factor 2. The subsampling factor of the output convolutional layer could be adjusted to adapt the changes in the input layer dimension, resulting in an adaptive implementation.

In a 1D CNN topology, following the step-by-step explanations in the reference article (Kiranyaz, Ince and Gabbouj, 2016), we may first write the 1D forward propagation from the previous convolution layer *l*−*1* to the input of *k*th neuron in the current layer *l* as,

$$x_k^l = b_k^l + \sum_{i=1}^{N_{l-1}} conv1D(w_{ik}^{l-1}, s_i^{l-1})$$

Equation 16. 1D convolution in a CNN layer.

where x_k^l represents the input, b_k^l is a scalar bias of the *k*th neuron at layer *l*, and s_i^{l-1} is the output of the *i*th neuron at layer *l*−*1*. w_{ik}^{l-1} is the kernel from the the *i*th neuron at layer *l*−*1* to the *k*th neuron at layer *l*. Then, the intermediate output of the neuron, y_k^l, can be written as follows:

$$y_k^l = f(x_k^l) \; and \; s_k^l = y_k^l \downarrow ss$$

Equation 17. Output of the activation function and subsampling in a CNN layer.

where s_k^l and "$\downarrow ss$" represent the output of the *k*th neuron of the layer *l*, and the subsampling operation with the factor *ss* respectively. In this equation, f is the activation function.

The backpropagation (BP) steps could be briefly summarized as follows. We start backpropagating the error from the output MLP layer. If *l*=1 for the input layer, *l*=*L* for the output layer, and N_L for the number of classes in the dataset are used, then the mean-squared error (*MSE*), E_P, in the output layer for the input vector *p*, its corresponding target $\boldsymbol{t}^p$, and output vector $[y_1^L, \ldots . y_{N_L}^L]$ can be expressed as follows:

$$E_P = MSE\left(\boldsymbol{t}^p, \left[y_1^L, \ldots . y_{N_L}^L\right]'\right) = \sum_{i=1}^{N_L} (y_i^L - t_i^p)^2$$

Equation 18. Mean-squared error at the output layer of a 1D CNN

To apply the gradient descent method, this error should be minimized using the derivatives of the error with respect to each weight (W_{ik}^{l-1}) and bias (b_k^l) connected to the corresponding neuron (i.e., *kth* neuron). We now need to compute the derivative of the error. All the delta errors can be calculated by backpropagating the error through the MLP layers and once we have them, we can update the weights and bias of each neuron by gradient descent method using the Equation 19. Therefore, the delta error of the *k*th neuron at layer *l* ($\Delta_k^l = \frac{\partial E}{\partial x_k^l}$) is used to update the bias of that neuron and all the weights of the neurons in the preceding layer connected to that neuron using the chain rule as,

$$\frac{\partial E}{\partial w_{ik}^{l-1}} = \Delta_k^l y_i^{l-1} \;\; and \;\; \frac{\partial E}{\partial b_k^l} = \Delta_k^l$$

Equation 19. Weight and bias sensitivities in the MLP layers of a 1D CNN.

Then, the regular (scalar) BP can be performed from the first MLP layer to the last CNN layer as,

$$\frac{\partial E}{\partial s_k^l} = \Delta s_k^l = \sum_{i=1}^{N_{l+1}} \frac{\partial E}{\partial x_i^{l+1}} \frac{\partial x_i^{l+1}}{\partial s_k^l} = \sum_{i=1}^{N_{l+1}} \Delta_i^{l+1} w_{ki}^l$$

Equation 20. BP from the first MLP layer to the last CNN layer in a 1D CNN.

When the first BP is initiated from the next layer *l+1*, to the current layer *l*, we can carry on the BP to the input delta, Δ_k^l, of the CNN layer *l*. If the zero order up-sampled map is given as, $\mathrm{us}_k^l = \mathrm{up}(\mathrm{s}_k^l)$, then we can compute the delta error as,

$$\Delta_k^l = \frac{\partial E}{\partial y_k^l}\frac{\partial y_k^l}{\partial x_k^l} = \frac{\partial E}{\partial us_k^l}\frac{\partial us_k^l}{\partial y_k^l} f'\left(x_k^l\right) = up(\Delta \mathrm{s}_k^l)\beta f\left(x_k^l\right)$$

Equation 21. The input delta, Δ_k^l, of the CNN layer l in a 1D CNN.

where $\beta = (ss)^{-1}$, as each element of s_k^l was obtained by averaging *l* number of elements of the intermediate output, y_k^l. The inter-BP (among CNN layers) of the delta error can now be written as,

$$\Delta \mathrm{s}_k^l = \sum_{i=1}^{N_{l+1}} conv1Dz(\Delta_i^{l+1}, rev\left(w_{ki}^l\right))$$

Equation 22. The inter-BP (among CNN layers) of the delta error in a 1D CNN.

where *rev(.)* reverses the array and *conv1Dz(.,.)* is used to perform full 1D convolution with *K*–1 zero padding.

The weight and bias gradient magnitudes are then computed from the following equation as,

$$\frac{\partial E}{\partial w_{ik}^l} = conv1D\left(s_k^l, \Delta_i^{l+1}\right) \ and \ \frac{\partial E}{\partial b_k^l} = \sum_n \Delta_k^l(n)$$

Equation 23. The weight and bias sensitivities of hidden convolutional layers in a 1D CNN.

Finally, after the weight and bias gradient magnitudes are calculated, they will be used to update weights and biases using the learning factor, ε as,

$$w_{ik}^{l-1}(t+1) = w_{ik}^{l-1}(t) - \varepsilon \frac{\partial E}{\partial w_{ik}^{l-1}} \;\; and \;\; b_k^l(t+1) = b_k^l(t) - \varepsilon \frac{\partial E}{\partial b_k^l}$$

Equation 24. Weight and bias update equations in a 1D CNN.

To get an in-depth knowledge of how BP algorithm works in a 1D CNN, one can refer to the paper by (Kiranyaz, Ince and Gabbouj, 2016). The iterative nature of BP used to train the 1D CNN classifier can be summarized as follows (Kiranyaz et al., 2021):

1) Initialize the weights and biases of the network randomly.

2) For each BP iteration DO:

a. For each training sample in the dataset, DO:

i. FP: Feed the training sample to the input layer and forward propagate towards the output layer to find outputs of each neuron at each layer, s_i^l, $\forall i \in [1, N_l]$, and $\forall l \in [1, L]$.

ii. BP: Compute the delta error at the output layer and carry on back-propagating it to first hidden layer to compute the input delta errors, Δ_k^l, $\forall k \in [1, N_l]$, and $\forall l \in [1, L]$.

iii. PP: Post-process to compute the weight and bias sensitivities using the Equation 23.

iv. Update: Update both the weights and biases of the network incrementally by the Gradient Descent update rule using the Equation 24.

(Eren, Ince and Kiranyaz, 2019) proposed a generic BFDD method based on adaptive 1D CNNs. The adaptive 1D CNN classifiers were evaluated on two benchmark datasets: Case Western Reserve University (CWRU) and Intelligent Maintenance System (IMS) bearing data. The raw vibration data were preprocessed to

use more compact model. Preprocessing involves down sampling and normalization stages. A 1D CNN model with 3 convolutional and 2 dense layers were used. The number of classes was five, and they are inner-race fault, ball fault, and three kinds of outer-race faults (located at 3 o'clock, 6 o'clock which is orthogonal to the load zone and 12 o'clock). In this paper, 10-fold cross-validation was implemented and the overall diagnosis accuracy of 93.2% was obtained on CWRU data with these settings.

3.5. Self-Organized Operational Neural Networks

Conventional CNN architecture is built upon the classical linear neuron model like MLPs, but it also introduces two additional constraints: weight sharing and kernel-wise limited connections. Therefore, these constraints introduced the convolution equation (Equation 16) utilized in CNNs. Several studies have recently showed that CNNs that are based on the first-order linear neuron model may not achieve a sufficient degree of learning if a sufficient network depth is not ensured (Kiranyaz et al., 2021). To achieve a high heterogeneity level, Self-Organized Operational Neural Networks (Self-ONNs) have been proposed by (Kiranyaz et al., 2021). Self-ONNs with minimal network complexity have been proven to maximize the learning performance when the training data is scarce, and some examples of the superior regression capability of Self-ONNs over image denoising, restoration and segmentation can be found in the following study (Kiranyaz et al., 2021). In this thesis**,** 1D Self-ONNs with generative neurons are also used for bearing fault severity classification on the mentioned benchmark datasets. The rest of this section briefly discusses 1D Self-ONNs and compare it to the 1D ONNs and CNNs. One can generalize the concepts for 2D Self-ONNs easily, and the further details can be found in the paper published by (Kiranyaz et al., 2021).

Firstly, we can again think of 1D 'same' convolution operation with unit stride and the required amount of zero padding. In a 1D CNN, the output of kth neuron in the layer l can then be written as follows:

$$x_k^l = b_k^l + \sum_{i=0}^{N_{l-1}} x_{ik}^l$$

Equation 25. The output of kth neuron of the layer l in a 1D CNN.

where b_k^l is the bias of the corresponding neuron, and x_{ik}^l is given as,

$$x_{ik}^l = Conv1D\left(w_{ik}, y_i^{l-1}\right) \; and \; x_{ik}^l(m) = \sum_{r=0}^{K-1} w_{ik}^l(r) y_i^{l-1}(m+r)$$

Equation 26. 1D convolution operation in CNN layers of a 1D CNN.

In Equation 26, $w_{ik} \in \mathbb{R}^K$ is the kernel connecting the *i*th neuron of *(l-1)*th layer to *k*th neuron of *l*th layer, while $x_{ik}^l \in \mathbb{R}^M$ is the input map, and $y_i^{l-1} \in \mathbb{R}^M$ is the *(l-1)*th layer's *i*th neuron's output.

The equation above can be generalized for an operational neuron as follows:

$$\overline{x_{ik}^l}(m) = P_k^l\left(\psi_k^l\left(w_{ik}^l(r), y_i^{l-1}(m+r)\right)\right)_{r=0}^{K-1}$$

Equation 27. The output of generalized operational neuron.

where $\psi_l^k(\cdot): \mathbb{R}^{M\times K} \to \mathbb{R}^{M\times K}$ and $P_k^l(\cdot): \mathbb{R}^K \to \mathbb{R}^1$ are called nodal and pool operators, respectively, assigned to the *k*th neuron of *l*th layer.

An optimal set of nodal ψ and pool P operators in a heterogenous ONN configuration could be searched iteratively from a potential set of operators using the Greedy Iterative Search (GIS) algorithm. Then, these operators are assigned to all neurons of the corresponding hidden layer to configure the final ONN. However, there are several drawbacks of conventional ONN architecture proposed in the

literature (Kiranyaz et al., 2020). The first issue is caused by the use of a single operator set for all neurons in a hidden layer, which limits heterogeneity. Secondly, hand-crafting a collection of possible operators and looking for the best one for each neuron result in a significant overhead. Finally, for the given learning problem, the right operator may not be expressed with well-defined functions, so it may not adapt or customize the operators**.**

To overcome these drawbacks, Self-ONNs with generative neurons were proposed. Self-ONNs have the capacity to self-organize network operators during training without the use of any operator set or a prior search process for optimal operators. Also, the use of a single nodal operator for all neurons in a hidden layer of an ONN is eliminated using the generative neuron concept in Self-ONN architecture. The core idea behind generative neurons is that each neuron may generate any combination of nodal operators, hence they do not have to be well-known functions such as linear, exponential and sinusoids. In Figure 12, 1D kernels of CNN, ONN and Self-ONN with generative neurons are shown. As seen in the figure, while the convolutional and operational neurons of the CNN and ONN architecture have fixed (static) nodal operators, Self-ONNs with generative neurons may produce any nodal operator Ψ for each kernel element during training.

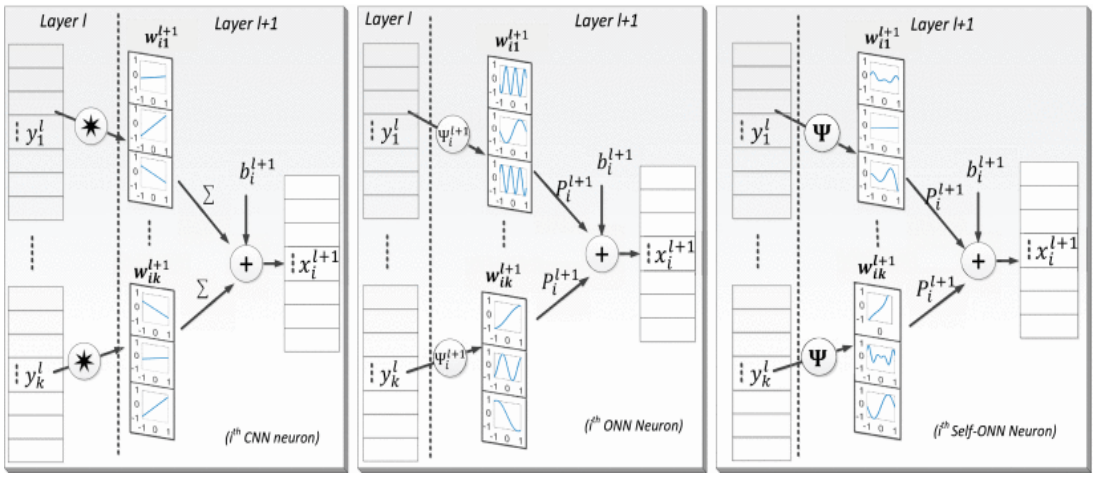

Figure 12. 1D nodal operations of the ith neuron of CNN (left), ONN (middle) and Self-ONN (right) (Source: Ince et al., 2021).

Nodal transformation in Self-ONNs can be formulated using the Taylor series function approximation. For an infinitely differentiable function $f(x)$, i.e., derivatives of all orders exist, Taylor series can be written about the point a as,

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Equation 28. Taylor series function approximation.

Then, we can take the Qth order approximation of the Equation 28, and write the Taylor polynomial as:

$$f(x)^{(Q,a)} = \sum_{n=0}^{Q} \frac{f^{(n)}(a)}{n!}x^n$$

Equation 29. Qth order Taylor series approximation.

This equation helps us to approximate any function $f(x)$ around the point a. During the BP training, the coefficients $\frac{f^{(n)}}{n!}$ are optimized at each iteration, which is also equivalent to customizing the nodal operator of each kernel element. As an example, if the neuron outputs are bounded by *tanh* activation function ([-1,1]), we can generate any transformation (perhaps nonlinear) near a point (midpoint 0) with the Qth order Maclaurin series**.** This is the underlying foundation of generative neurons in Self-ONNs. Nodal transformation of a generative neuron can be more specifically written in the following general form as,

$$\widetilde{\psi}_k^l\left(w_{ik}^{l(Q)}(r), y_i^{l-1}(m+r)\right) = \sum_{q=1}^{Q} w_{ik}^{l(Q)}(r,q)\left(y_i^{l-1}(m+r)\right)^q$$

Equation 30. The general form of nodal transformations in a generative neuron.

where Q is the degree of Taylor polynomial and $w_{ik}^{l(Q)}$ is a learnable kernel of the network. The $K \times 1$ kernel vector w_{ik}^{l} in 1D CNN topology has now been replaced by a $K \times Q$ matrix $w_{ik}^{l(Q)} \in \mathbb{R}^{K \times Q}$ in Self-ONNs, and this matrix is basically formed by replacing each element $w_{ik}^{l}(r)$ with a Q-dimensional vector $w_{ik}^{l(Q)}(r) = [w_{ik}^{l(Q)}(r,0), w_{ik}^{l(Q)}(r,1), \dots, w_{ik}^{l(Q)}(Q-1)]$ (Ince et al., 2021). Furthermore, it can be seen from this equation that $\widetilde{\psi_k^l}$ is not a fixed operator for each individual output y_i^{l-1}, and it enforces Q times more parameters than the CNN model. Then, the input map of the generative neuron $\tilde{x}_{ik}^{l}$ could be further written as:

$$\widetilde{x_{ik}^{l}}(m) = P_k^l \left(\sum_{q=1}^{Q} w_{ik}^{l(Q)}(r,q) \left(y_i^{l-1}(m+r) \right)^q \right)_{r=0}^{K-1}$$

Equation 31. The input map of generative neuron $\tilde{x}_{ik}^{l}$.

Consequently, Self-ONNs has the following advantages compared to their counterparts CNNs and ONNs. First of all, there is no need for the search of optimal operator in each neuron connection from a set of well-defined functions since Self-ONNs are able to self-organize the network operators with the generative neurons during training. Secondly, the heterogeneity is not restricted to a single nodal operator for all kernel connections as in the case of ONNs, thus each neuron will rather be addressed by the generative neurons. Lastly, along with these advantages, unlike ONNs, a Self-ONN layer can be parallelized more efficiently (Kiranyaz et al., 2021). The forward and back-propagation formulation in a Self-ONN neuron can be studied in detail from (Kiranyaz et al., 2021).

(Ince et al., 2021) first proposed 1D Self-ONNs for bearing fault severity classification. The 1D Self-ONN model having three operational and two dense layers was evaluated on NASA/IMS bearing data. Both x and y-axis acceleration waveforms were inputted to the model for inner-race and ball fault severity classification. For inner-race fault severity classification, Self-ONNs had around 3-8% F1 gains over the

1D CNN with the same configuration in all metrics. On the other hand, for ball faults, Self-ONNs had slightly lower F1 performance gains (2-4%) over the CNN.

CHAPTER 4: 1D CNN AND SELF-ONN PERFORMANCE RESULTS ON BENCHMARK DATASETS

4.1. Datasets

The performance of machine learning models largely relies on the quality and quantity of data used to train the model; thus, a good collection of data is required. For BFDD problem, data can be collected from electric motors with artificially induced bearing faults or run-to-failure tests can be performed to simulate the natural degradation of bearings. Different parameters such as stator current and motor vibration may also be used for BFDD, but in this thesis, the focus is on motor vibration since the early signs appear in the vibration data. There are several public bearing fault datasets such as CWRU, NASA/IMS, Paderborn University, FEMTO, MFPT and University of Ottawa's bearing variable speed data, and they allow us to evaluate machine learning algorithms for the BFDD problem. In this thesis, CWRU and University of Ottawa's bearing variable speed dataset (both utilizing motor vibration), are used to evaluate 1D CNN and Self-ONNs models with raw vibration data.

4.1.1. CWRU Dataset

Case Western Reserve University (CWRU) bearing data is an open-source dataset that can be easily accessed from the web page of CWRU bearing data center (CWRU, 2004). The website presents accelerometer data collected from a 2-hp electric motor for healthy and faulty ball bearings. Figure 13 depicts the CWRU motor test bench with a 2-hp electric motor (left), a torque transducer/encoder (center), and a dynamometer (right). Torque is delivered to the rotating shaft by a dynamometer.

Electro-discharge machining (EDM) was used to induce defects into the bearings. To simulate different bearing fault severity levels, faults are generated at the outer raceway, ball and inner raceway in different diameters ranging from 0.007 inches to 0.040 inches. Accelerometer data was gathered using three accelerometers mounted

onto the fan-end, drive-end, and the base of motor housing under varied motor loading conditions (0 to 3 hp) for healthy bearings, and the bearings with single-point fan-end and drive-end defects. As a result of variable loading, motor speed changed slightly from 1797 rpm to 1720 rpm.

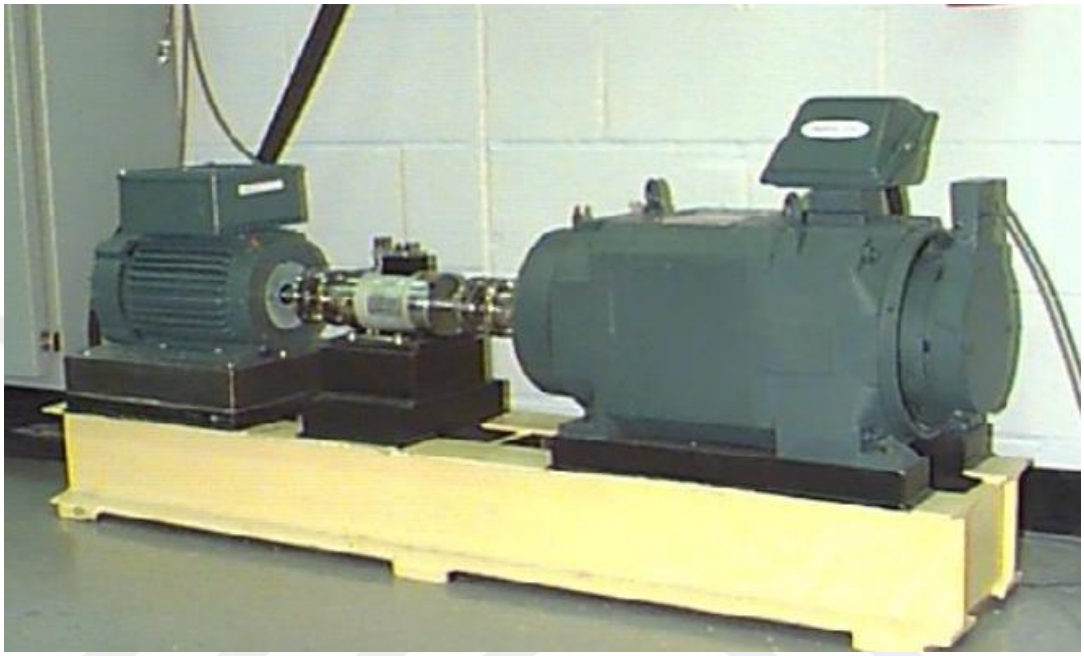

Figure 13. CWRU dataset motor bearing test platform.

All the data files, i.e. 161 records, were stored in .mat format, and the dataset was divided into four groups as 12k drive-end bearing fault, 12k fan-end bearing fault, 48k normal-baseline and 48k drive-end bearing fault data. 12k and 48k specifies the sampling frequency used to record the vibration data. In each group, the data can further be divided into subclasses according to motor load, fault diameter and the location of the bearing fault. Furthermore, since outer-race bearing faults are stationary, the location of the fault relative to the load zone of the bearing has an impact on the vibration response. To take this effect into account, data was collected for outer race faults located at 3 o'clock (directly in the load zone), at 6 o'clock (orthogonal to the load zone), and at 12 o'clock. The data files were named as follows. The first two letters represent the fault position, the next three numbers give the fault diameter in inches in the format 0.XXX", and the last digit represents the bearing load. For example, the data file IR007_0 was collected from a 0.007 inches inner race bearing

fault induced motor under no load (0 hp) condition. For outer race faults, @ symbol is also used to indicate the fault location relative to the load zone.

Table 1 gives an information about the bearings used in drive-end and fan-end of the electric motor. Using the bearing geometry, the corresponding bearing fault frequencies were calculated using Equation 1 through 4, and they were shown in Table 2. Number of balls is 9, and 8 for 6205 and 6203 SKF ball bearings, respectively.

Table 1. CWRU dataset fan-end and drive-end bearing information.

Bearing Type	**Inside Diameter (mm)**	**Outside Diameter (mm)**	**Thickness (mm)**	**Ball Diameter (mm)**	**Pitch Diameter (mm)**
6205-2RS JEM SKF (Drive-end)	25	52	15	7.94	39.04
6203-2RS JEM SKF (Fan-end)	17	40	12	6.75	28.50

Table 2. CWRU dataset fan-end and drive-end bearing fault frequencies.

Bearing Type	**Bearing fault frequencies in Hz (Multiple of running speed)**			
	BPFI	**BPFO**	**BSF**	**FTF**
6205-2RS JEM SKF (Drive-end)	5.42	3.58	4.71	0.40
6203-2RS JEM SKF (Fan-end)	4.95	3.05	3.99	0.38

4.1.2. University of Ottawa's Variable Speed Bearing Dataset

Since bearings usually operate under variable motor speed conditions, it is crucial to test the performance of algorithms under time-varying speed conditions. University of Ottawa's Variable Speed Bearing Dataset made available in Mendeley data (Huang and Baddour, 2019) and it includes accelerometer data recorded from motor bearings with various health conditions under variable motor speed. The health conditions of the bearings are healthy, outer-race defect, inner-race defect, ball defect, and faulty with combined defects. Therefore, there are five different classes to be classified in this dataset. Vibration signals are collected for decreasing operating speed, increasing speed, decreasing then increasing speed and increasing then decreasing speed conditions.

The experimental setup is shown in Figure 14. The rotational speed of the motor shaft was adjusted by an AC drive. Two ball bearings were used to support the shaft. In Figure 14, the bearing on the right is the experimental bearing and the one on the left is healthy. The experimental bearing was replaced by the bearing with the mentioned health conditions, and the accelerometer data along with motor shaft speed were recorded.

There are 60 data files in this dataset. 3 samples were collected for one experimental setting. Each data file (.mat file) consists of two channels. Channel 1 includes the vibration signal measured using the accelerometer and Channel 2 stores the rotational speed data measured using the encoder. In each file, the sampling frequency for both channels is 200 kHz and the sampling duration is 10 seconds. Figure 15 shows how the dataset is numbered.

The details of the data files numbered in Figure 15 can be found in (Huang and Baddour, 2019). For example, the data file I-A-2 includes the accelerometer data gathered from a bearing with an inner-race fault and the operating rotational speed is increased from 13.0 Hz to 25.7 Hz.

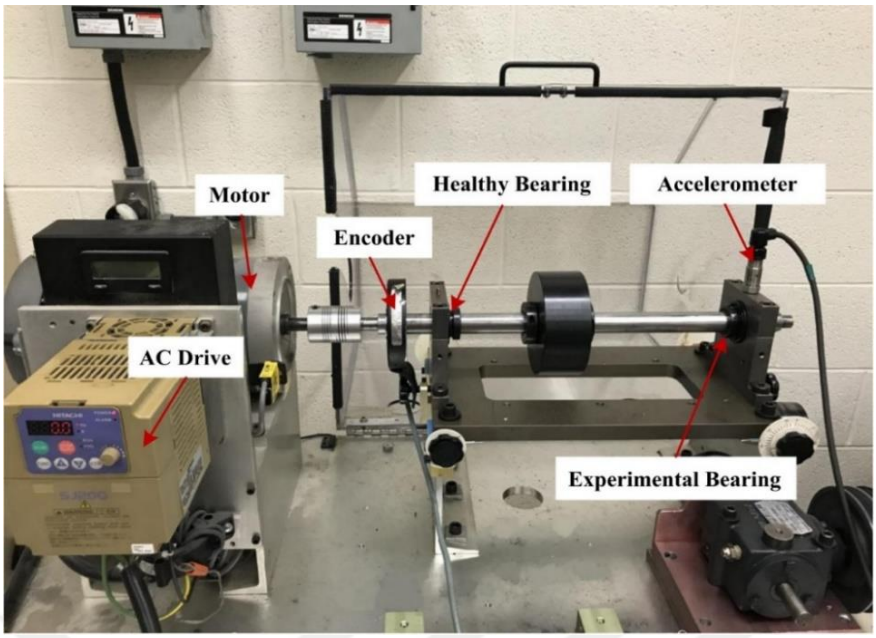

Figure 14. Experimental setup for University of Ottawa's Bearing Dataset.

Bearing health conditions	Speed varying conditions			
	Increasing speed	Decreasing speed	Increasing then decreasing speed	Decreasing then increasing speed
Healthy	H-A-1	H-B-1	H-C-1	H-D-1
	H-A-2	H-B-2	H-C-2	H-D-2
	H-A-3	H-B-3	H-C-3	H-D-3
Faulty (inner race fault)	I-A-1	I-B-1	I-C-1	I-D-1
	I-A-2	I-B-2	I-C-2	I-D-2
	I-A-3	I-B-3	I-C-3	I-D-3
Faulty (outer race fault)	O-A-1	O-B-1	O-C-1	O-D-1
	O-A-2	O-B-2	O-C-2	O-D-2
	O-A-3	O-B-3	O-C-3	O-D-3
Faulty (ball fault)	B-A-1	B-B-1	B-C-1	B-D-1
	B-A-2	B-B-2	B-C-2	B-D-2
	B-A-3	B-B-3	B-C-3	B-D-3
Faulty (combination of faults)	C-A-1	C-B-1	C-C-1	C-D-1
	C-A-2	C-B-2	C-C-2	C-D-2
	C-A-3	C-B-3	C-C-3	C-D-3

Figure 15. University of Ottawa's bearing dataset numbering.

4.2. CWRU Dataset 1D CNN and Self-ONN Results

12k drive-end bearing fault and normal baseline datasets were used to classify drive-end bearing fault severity levels. Among the drive-end, fan-end, and base accelerometer data, for this scenario, drive-end accelerometer data was used. Sample faulty and healthy motor vibration signals and their amplitude spectrum are shown in Figure 17 through Figure 22. The 12k drive-end bearing dataset includes four different bearing health conditions in terms of the location of the fault: normal, outer-race fault, inner-race fault, and ball fault. Each fault type can be further divided into groups according to the fault diameters as 0.007 inches, 0.014 inches and 0.021 inches. Thus, we get a total of 10 classes to be classified as shown in Table 3.

Table 3. 10 different classes in CWRU dataset.

Class label	0	1	2	3	4	5	6	7	8	9
Fault location and fault size (mils)	Normal 0	IR 7	IR 14	IR 21	OR 7	OR 14	OR 21	BF 7	BF 14	BF 21

Since the raw vibration data was collected under different loading conditions, we can test the performance of our ML models for two different cases. In the first case, we can both train and test a model under a single loading condition, and as a second scenario, we can test the performance of a model across different load domains. For example, one can train a model under no load condition, and test its performance under the load condition of 1 hp. For this purpose, the whole dataset was grouped into sub-datasets according to the loading condition of the motor with each dataset having 10 different classes as given in Table 4.

Table 4. CWRU sub-datasets according to loading conditions (Number of training / validation / test samples).

Fault location and fault size (mils)	Dataset A (0 hp)	Dataset B (1 hp)	Dataset C (2 hp)	Dataset D (3hp)	Dataset E (0/1/2/3 hp)
Training (Number of samples)	3186	3304	3304	3304	13098
Validation (Number of samples)	1593	1652	1652	1652	6549
Test (Number of samples)	810	840	840	840	3330

In each dataset, to compare the performance of 1D CNN and Self-ONN architectures, each data file was divided into 4 equal pieces without any shuffling. The last piece was used as holdout data for testing purposes. The non-holdout data was then split into three equal pieces. Two of them were used for training in each fold and one for validation to apply 3-fold cross-validation as shown in Figure 16.

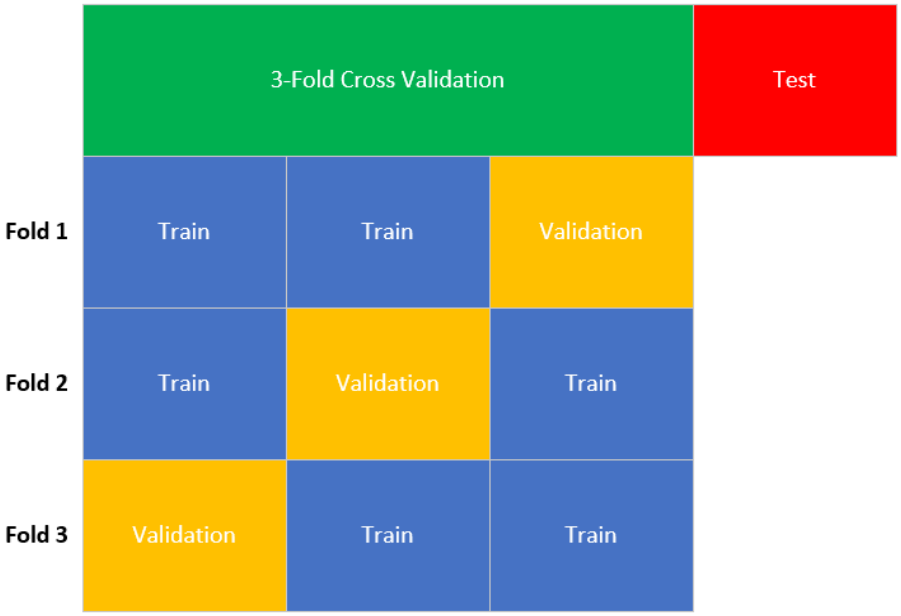

Figure 16. 3-fold cross validation and holdout (test) data.

For each model, the input window size was chosen as 500 time-domain samples so that the raw vibration includes at least one revolution of the motor shaft. The training and validation samples were augmented by slicing the raw vibration data with 50% overlap (250 time-domain samples), and no overlap was used for the testing samples. After the segmentation and data augmentation process, min-max normalization was applied on the input data using the Equation 32. Min-max normalization rescales the data to fall within the range [−1, 1].

$$x_{scaled} = 2 \times \left(\frac{x - x_{min}}{x_{max} - x_{min}} \right) - 1$$

Equation 32. Min-max normalization for the target range [−1, 1].

As a result, datasets A has now 3,186 training, 1,593 validation and 810 testing samples, while datasets B, C and D each contains 3,304 training, 1,532 validation and 840 testing samples for ten different bearing health conditions. On the other hand, dataset E contains all four loads (0/1/2/3 hp) with 13,098 training, 6,549 validation and 3,330 test samples.

3 different BP runs were performed for each fold, and the maximum number of epochs were chosen as 40. For the corresponding fold, among all BP runs and epochs, the model with the minimum validation loss was chosen as the best-performing model, then this model was used for testing. The performance result of this model on the test dataset was then reported for the corresponding fold. By this way, the average of 3 folds can be taken to indicate the average fault classification result.

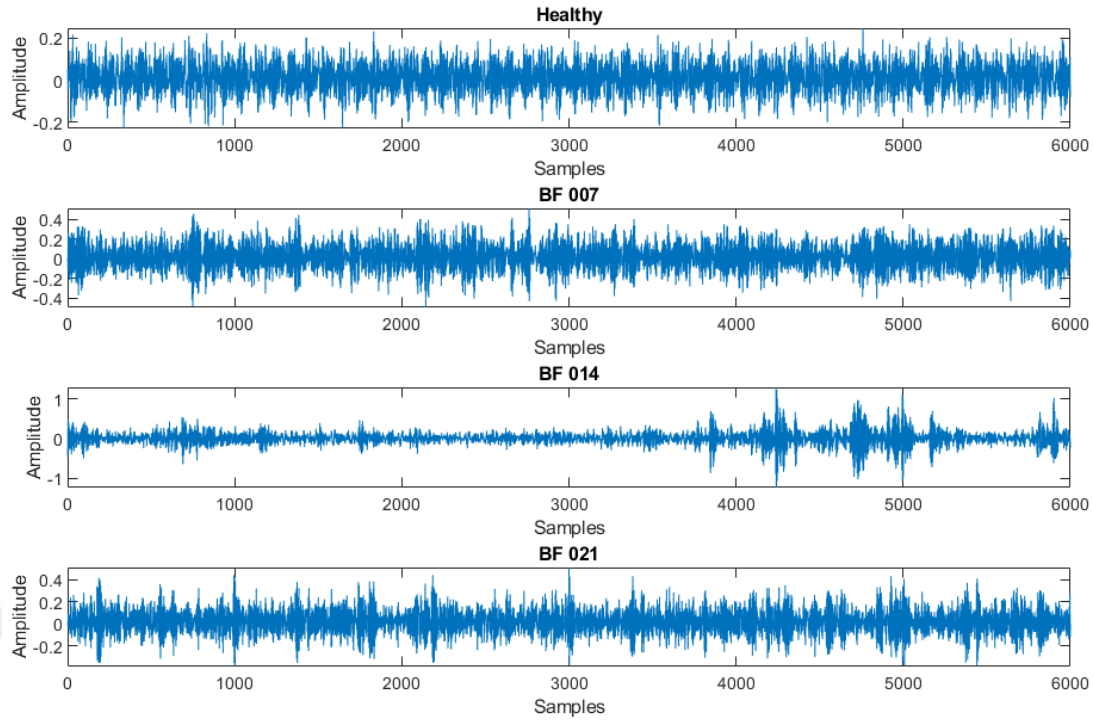

Figure 17. Sample vibration waveforms for healthy and ball fault conditions in time domain.

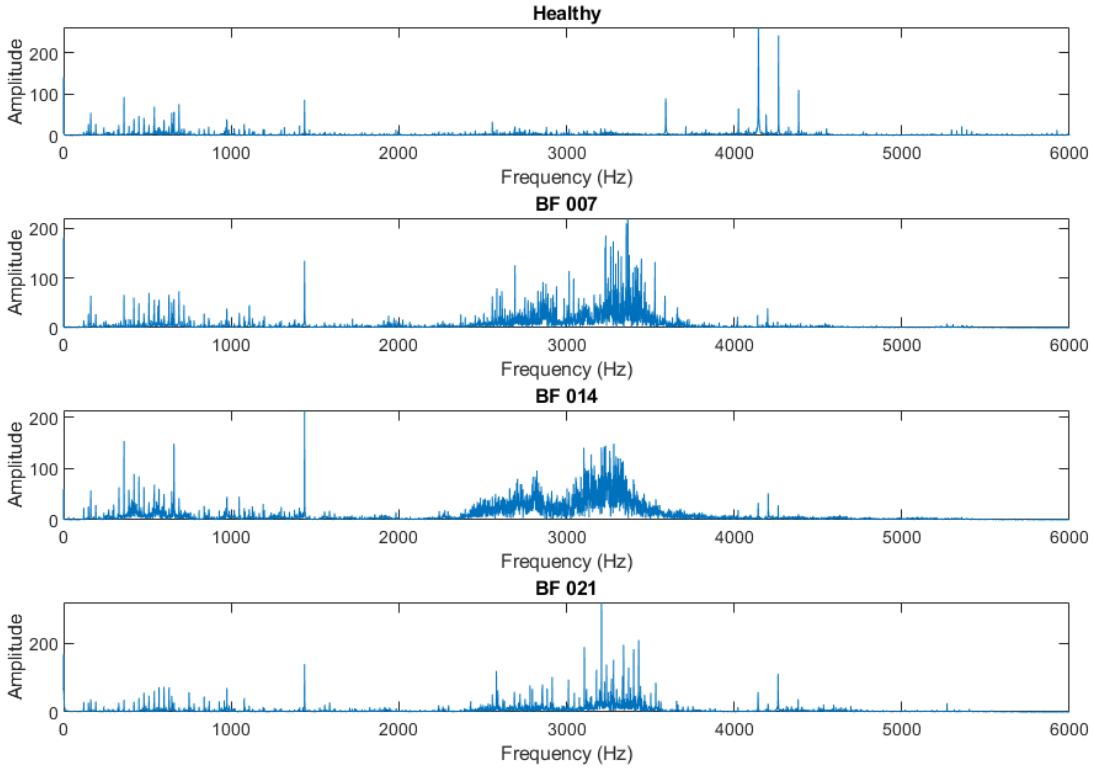

Figure 18. Amplitude spectrum of vibration signals for healthy and ball fault conditions.

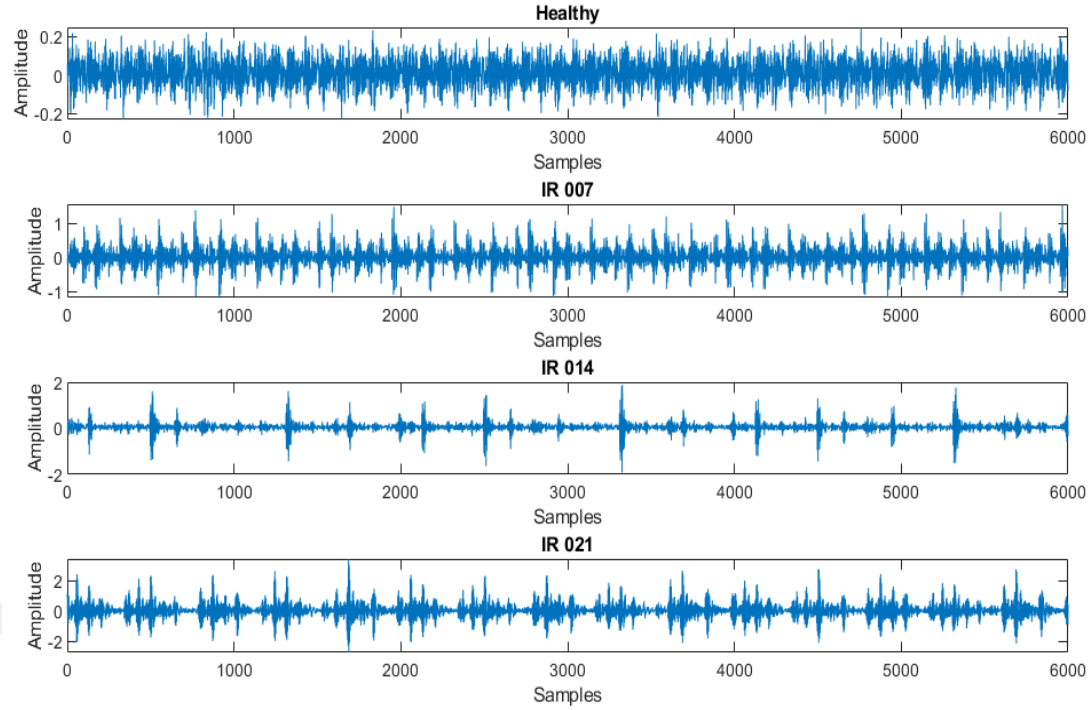

Figure 19. Sample vibration waveforms for healthy and inner-race fault conditions in time domain.

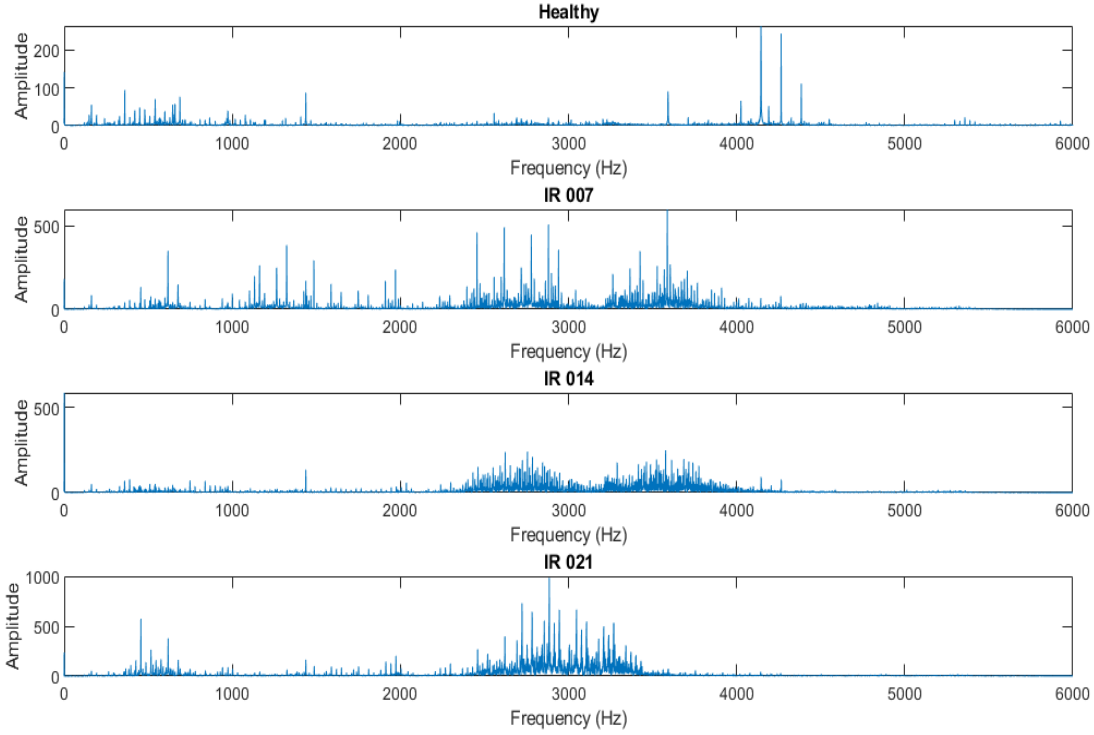

Figure 20. Amplitude spectrum of vibration signals for healthy and inner-race fault conditions.

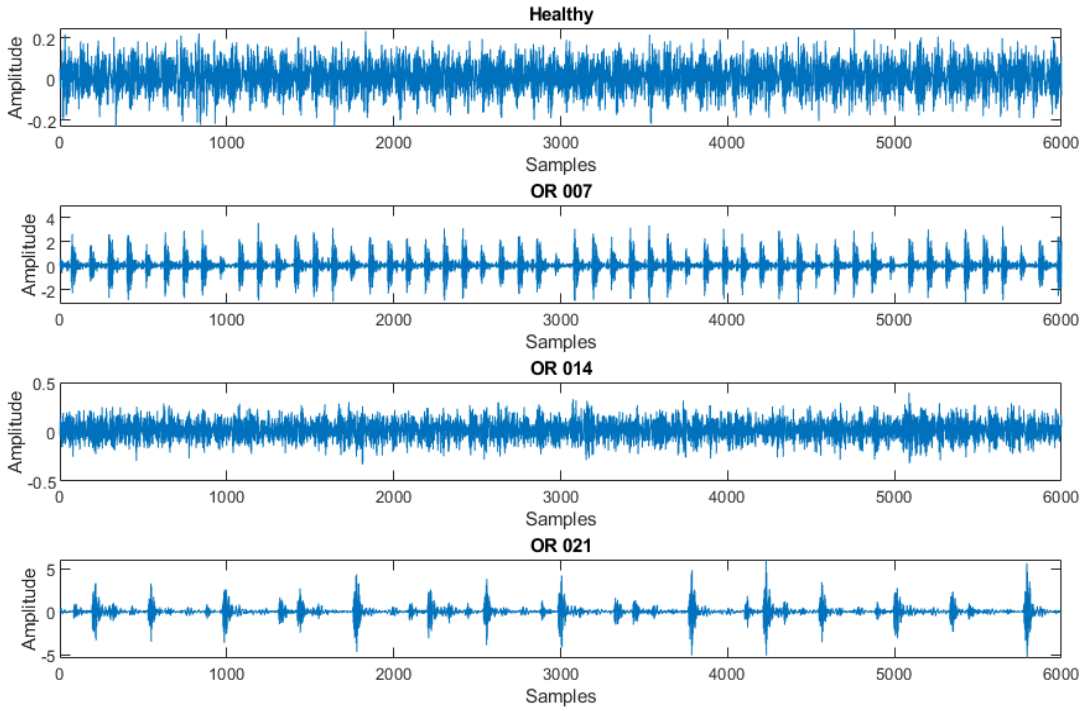

Figure 21. Sample vibration waveforms for healthy and outer-race fault conditions in time domain.

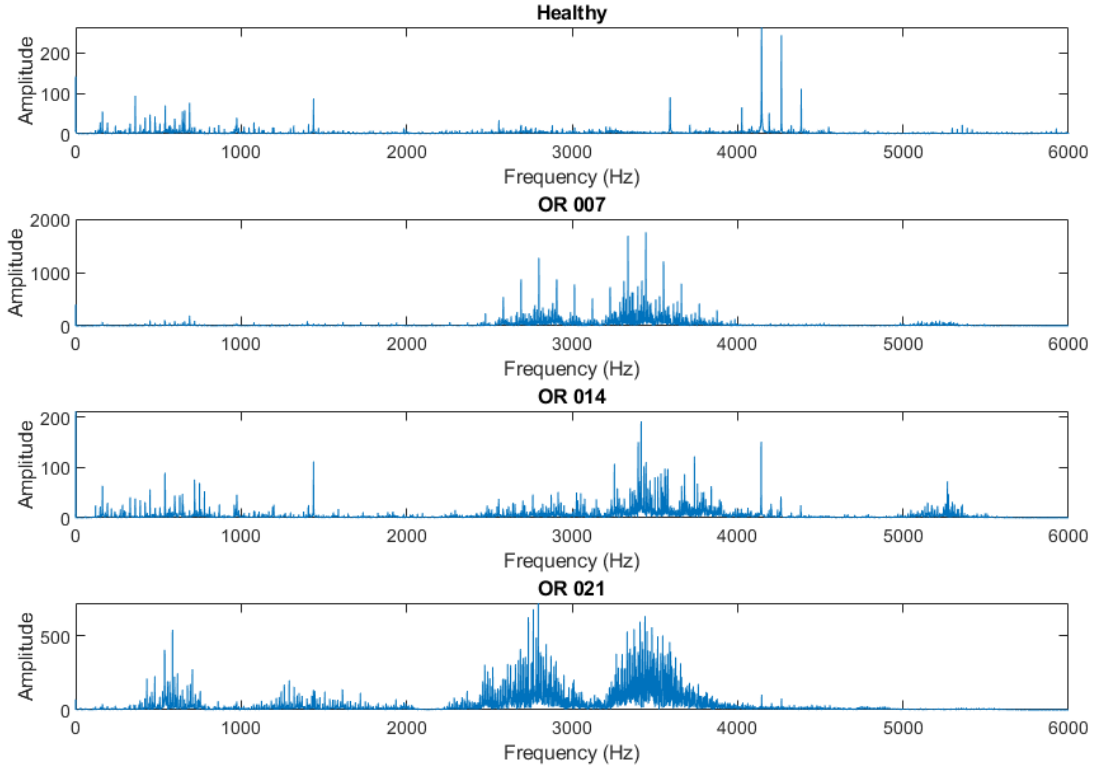

Figure 22. Amplitude spectrum of vibration signals for healthy and outer-race fault conditions.

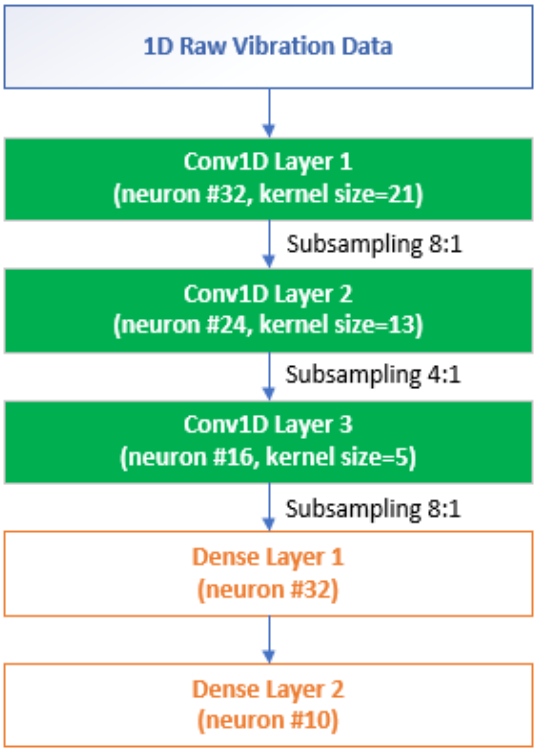

Figure 23. 1D CNN classifier with three Conv1D (number of neurons, kernel size) and two dense layers (number of neurons).

The datasets A to E were first used to assess the performance of the proposed 1D CNN model for a single working load condition. For example, all the training, validation and test samples were from dataset A, when the performance of the model was evaluated under no load condition (0 hp). For a single operating condition, Self-ONNs was not used, since this classification task is an easy problem which can be handled by 1D CNNs. The result will also show the effectiveness of 1D CNNs for a single operating condition.

The 1D CNN model given in Figure 23 was used for each dataset. It has 3 convolutional and 2 dense layers. Normalized raw vibration data with size 500 (time-domain samples) was inputted to the model. The 1D convolutional layers has 32, 24, and 16 neurons with kernel sizes 21, 13 and 5, respectively. There are 32 neurons in the hidden MLP layer, and the output dense layer size is 10 which is equal to the number of classes. At the output MLP layer, cross-entropy loss with SoftMax function

was utilized. The hyperbolic tangent activation function *tanh* was used through all convolutional and MLP layers. The subsampling factors (max-pooling) for convolutional layers were selected as 8, 4 and 8 respectively. The Adam optimizer with a learning rate of 0.001 was used. The batch size was chosen as 32 for each model.

To evaluate the performance of each model, the most widely used performance metrics accuracy, recall, precision, and F1-score were used. Recall is defined as the ratio of correctly predicted positive observations to all observations in actual class. On the other hand, precision is the ratio of correctly predicted positive observations to the total predicted positive observations. In many cases, we would like to summarize the performance with a single number called F1-score, which can be defined as the harmonic mean of precision and recall. Also, accuracy is defined as the ratio of the number of correct predictions to the total number of predictions. One can formulate these performance metrics using false negatives (FN), false positives (FP), true negatives (TN) and true positives (TP) as follows:

$$Recall = \frac{TP}{TP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$F1\ Score = \frac{2 * Precision * Recall}{Precision + Recall}$$

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Equation 33. Precision, recall, F1-score and accuracy.

The drive-end bearing fault location and severity level classification performance results of the proposed 1D CNN model for datasets A (0 hp), B (1 hp), C (2 hp) and D (3 hp) are given in Table 5, and the result for dataset E is shown in Table 6. In terms of classification accuracy, the 1D CNN model with raw vibration data is compared with several machine learning models in Table 7. The results show that the proposed 1D CNN achieves comparable and even better results for some datasets,

although it uses a few thousand parameters. (Du et al., 2014) used wavelet leaders multifractal features to train an SVM under no load condition (dataset A), and the reported classification accuracy was 89.1%. The proposed 1D CNN model achieves 99.88% accuracy on the same dataset. (Jin et al., 2014) implemented trace ratio linear discriminant analysis (TR-LDA) for dimension reduction and fault diagnosis under 3 hp loading (dataset D), and the diagnosis accuracy was 92.5%. (Ding and He, 2017) proposed energy-fluctuated multiscale feature mining approach based on wavelet packet energy (WPE) image and deep convolutional networks and this method achieved classification accuracies of 98.8, 98.8, 99.4 and 99.4% for dataset A, B, C and D, respectively. However, this architecture is more complex than the proposed 1D CNN and requires WPE images for the input. (Wang et al., 2020) used a signal to image spatial transform method to generate 2D gray images from raw vibration data, and then implemented multi-head attention-based CNN to diagnose bearing faults. 5 different CNN models were designed, and the proposed CNN-E architecture had better generalization ability with an accuracy over 99% for each sub-dataset. However, this approach also requires a complex model with 455,210 trainable parameters, while the proposed 1D CNN has only 13,522 trainable parameters. Furthermore, the signal to image conversion required at the input of the model is a time-consuming task. (Chen, Zhang and Gao, 2021) used the raw vibration data as input and using two CNNs with different kernel sizes, different frequency signal characteristics were extracted. After that, long short-term memory (LSTM) was used to identify the fault type using the extracted features. The MCNN-LSTM method obtained 98.46% accuracy under 3 hp loading condition.

Table 5. 1D CNN results for each sub-dataset (A, B, C, D) of CWRU bearing data.

Drive-end Bearing 1D CNN 3-Fold Cross Validation Average Performance Results						
Dataset	**A**			**B**		
Scores	**Precision**	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Normal	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
IR_007	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
IR_014	0.9890	1.0000	0.9945	1.0000	1.0000	1.0000
IR_021	1.0000	1.0000	1.0000	0.9890	1.0000	0.9945
OR_007	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
OR_014	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
OR_021	1.0000	0.9963	0.9981	1.0000	1.0000	1.0000
BF_007	1.0000	1.0000	1.0000	1.0000	0.9778	0.9888
BF_014	0.9945	1.0000	0.9972	1.0000	1.0000	1.0000
BF_021	1.0000	0.9944	0.9972	0.9890	1.0000	0.9945
Accuracy			**0.9988**			**0.9984**
Dataset	**C**			**D**		
Scores	**Precision**	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Normal	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
IR_007	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
IR_014	1.0000	0.9944	0.9972	1.0000	1.0000	1.0000
IR_021	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
OR_007	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
OR_014	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
OR_021	0.9982	1.0000	0.9991	1.0000	1.0000	1.0000
BF_007	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
BF_014	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
BF_021	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Accuracy			**0.9996**			**1.0000**

Table 6. 1D CNN results for dataset E of CWRU bearing data.

Dataset	**E**		
Scores	**Precision**	**Recall**	**F1 Score**
Normal	1.0000	1.0000	1.0000
IR_007	1.0000	1.0000	1.0000
IR_014	0.9986	0.9986	0.9986
IR_021	1.0000	1.0000	1.0000
OR_007	0.9995	1.0000	0.9998
OR_014	1.0000	1.0000	1.0000
OR_021	0.9995	0.9995	0.9995
BF_007	1.0000	0.9986	0.9993
BF_014	1.0000	1.0000	1.0000
BF_021	1.0000	1.0000	1.0000
Accuracy			**0.9997**

(Zhang et al., 2020) proposed a CNN with two dropout and two fully-connected layers (DFCNN). The raw vibration waveform was first converted into an image by filling the pixels of the image using the time-domain signals. Then, these images were fed into DFCNN model. The DFCNN model utilizes wide kernels in the first convolutional layer, and the kernel size was reduced in the following layers. To improve the generalization capability, two dropout layers were used. The average accuracy of DFCNN model is also given in Table 7.

Table 7. Comparison of different methods in terms of classification accuracy on CWRU bearing datasets.

Methods	**Dataset A**	**Dataset B**	**Dataset C**	**Dataset D**	**Dataset E**
Multifractal+SVM	89.1	-	-	-	-
TR-LDA	-	-	-	92.5	-
WPE+CNN	98.8	98.8	99.4	99.4	-
CNN-E	99.4	99.4	99.8	100	-
MCNN-LSTM	-	-	-	98.46	-
DFCNN	100	100	100	100	99.8
1D CNN	**99.88**	**99.84**	**99.96**	**100**	**99.97**

The performance of 1D CNNs (q=1) and Self-ONNs (q=3, 5 and 7) across different load domains were also investigated using CWRU bearing data. The 1D Self-ONN model shown in Figure 24 was used for this scenario. Number of epochs was reduced to 20 to avoid overfitting. 3 runs were performed for each case and the Adam optimizer with a learning rate of 0.001 was used. The batch size was again 32. For each source and target pair, one was used for training and validation, and the other for testing. For example, for A→B source and target pair, dataset A was used for training and validation. It was split into 3 equal pieces and 3-fold cross validation was performed on this data. Then, at each fold, the model with the minimum validation loss was used to evaluate the performance on the test dataset (dataset B). The average performance for 3 folds is presented in Table 8. In this table, 1D CNN (*2) shows the classification accuracy when the neurons in convolutional layers of 1D CNN are doubled from 32 to 64, and 24 to 48, 16 to 32, respectively. As we can see, 1D Self-ONN model with q=7 has the highest average classification accuracy with 91.90%. On the other hand, 1D CNN model has the average accuracy of 89% across all load domains. When we double the number of convolutional neurons for 1D CNN architecture, the average accuracy increases to 91.24%, but it is still less than the accuracy reached by the 1D Self-ONN model with q=7. This demonstrates the superiority of the 1D Self-ONNs over the CNNs for this problem. Furthermore, we can interpret from the results that when the speed difference of source and target load domains is small, we can obtain a better classification accuracy.

The classification accuracy of the proposed 1D Self-ONN model was also compared to some advanced architectures such as the Deep Convolutional Transfer Learning Network (DCTLN) (Guo et al., 2019), Deep Convolutional Neural Networks with Wide First-layer Kernels (WDCNN) (Zhang et al., 2017), Domain-Adversarial Training of Neural Networks (DANN) (Ganin et al., 2016), Discriminative Adversarial Domain Adaptation (DADA) (Tang and Jia, 2019) and an autoencoder with attention mechanism and 1D CNN LSTM classifier (Jang and Cho, 2021) in Table 9. The superiority of the proposed 1D Self-ONN model is again obvious from these results. DCTLN includes two main parts which are condition recognition and domain adaptation. The condition recognition module is constructed using a 1D CNN to automatically learn features. The domain adaptation module allows the 1D CNN to

learn domain-invariant features by maximizing domain recognition errors and minimizing the probability distribution distance. On the other hand, WDCNN takes raw vibration data as input and utilizes wide kernels in the first convolutional layer for extracting features and suppressing high frequency noise and small convolutional kernels in the preceding layers. Adaptive Batch Normalization (AdaBN) proposed by (Li et al., 2018) was used to enhance the domain adaptation ability of WDCNN model.

Furthermore, in DANN, gradient reversal layer is added to the conventional feed-forward neural network architecture. As the training progresses, the model extracts discriminative features for the main learning task on the source domain and indiscriminate with respect to the shift between source and target domains. Discriminative Adversarial Domain Adaptation (DADA) includes a feature extractor and an integrated category and domain classifier. This model encourages a mutually inhibitory relation between its domain prediction and category prediction for any input instance. Finally, the last method is composed of an attentional autoencoder for latent vector representation and a 1D CNN LSTM-based classifier to classify bearing failures from latent vectors. For domain adaptation, a feature space transformation was added to the model. The performance results of all these methods on CWRU data were obtained from the study conducted by (Jang and Cho, 2021). The detailed parameters for each model can be found in their study as well.

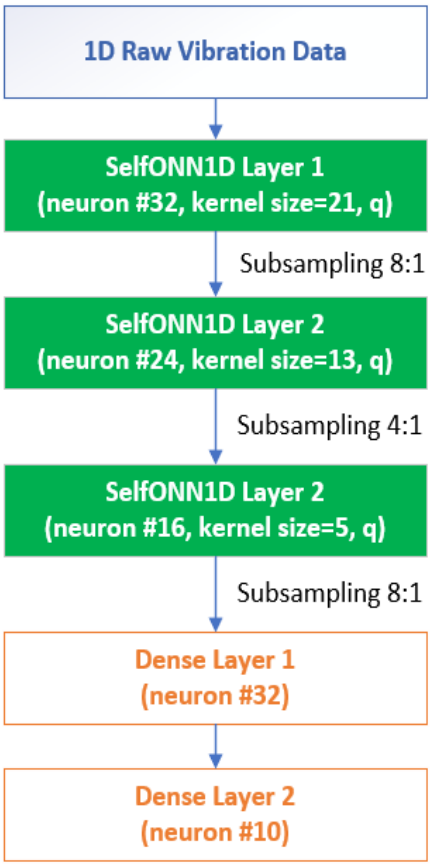

Figure 24. 1D Self-ONN classifier with three SelfONN1D layers (number of neurons, kernel size, the degree of the Taylor approximation) and two dense layers (number of neurons).

Table 8. 1D CNN and Self-ONN classification accuracies across different load domains.

Datasets Source→Target	**1D CNN**	**1D SelfONN (Q=3)**	**1D SelfONN (Q=5)**	**1D SelfONN (Q=7)**	**1D CNN (*2)**
A→B	91.71	94.42	93.05	93.44	**96.46**
A→C	88.12	91.31	**92.53**	91.40	90.10
A→D	75.56	79.66	**83.71**	80.57	78.30
B→A	95.17	95.62	93.34	**96.29**	93.79
B→C	98.28	98.81	98.88	98.33	**99.11**
B→D	83.94	90.19	**95.41**	91.55	92.66
C→A	93.10	**95.33**	94.50	94.24	95.21
C→B	94.12	96.53	94.57	96.16	**96.69**
C→D	93.57	**97.76**	95.14	96.66	97.08
D→A	83.10	85.35	84.43	**85.82**	83.17
D→B	83.06	84.56	83.72	**86.21**	83.80
D→C	88.32	90.07	89.40	**92.13**	88.49
AVG	89.00	91.63	91.56	**91.90**	91.24
Number of trainable parameters	13,522	38,674	63,826	88,978	50,490

Table 9. Comparison of different methods (Jang and Cho, 2021) in terms of classification accuracy across different load domains.

Task Source→Target	**1D SelfONN (Q=7)**	**Attentional Autoencoder + 1D CNN LSTM**	**DANN**	**DADA**	**DCTLN**	**WDCNN**
A→B	93.44	84.43	67.76	63.27	56.23	71.47
A→C	91.40	86.43	68.96	62.86	57.78	72.87
A→D	80.57	85.67	69.81	66.97	54.11	71.90
B→A	96.29	84.67	66.73	67.82	52.56	69.91
B→C	98.33	85.35	64.96	70.95	56.67	67.48
B→D	91.55	82.43	69.65	68.93	58.12	67.58
C→A	94.24	83.34	59.70	62.99	59.12	68.88
C→B	96.16	82.45	64.40	59.55	54.32	70.15
C→D	96.66	82.21	69.82	59.62	52.13	65.83
D→A	85.82	81.24	58.62	57.89	57.45	68.14
D→B	86.21	80.67	57.41	62.53	58.23	65.17
D→C	92.13	80.67	58.98	59.90	46.12	69.90
AVG	91.90	83.63	66.15	63.61	55.24	69.11

4.3. University of Ottawa's Bearing Dataset 1D CNN and Self-ONN Results

This dataset differs from the CWRU bearing data in that each sample was gathered for time-varying rotational speed conditions. The whole data was split into 3 sub-datasets: dataset X, dataset Y and dataset Z as shown in Figure 25. Each sub-dataset includes all the bearing health and speed varying conditions. The main goal was to diagnose bearing faults under time-varying rotational speed, so the number of classes are 5 and these are healthy, ball fault, inner-race fault, outer-race fault, and combined faults. Each sub-dataset can be used for training, validation or testing purposes, thus there are 6 cases in total as shown in Table 10.

The sampling frequency for this dataset is 200 kHz, thus it requires a very complex model if the raw vibration is inputted directly. To avoid this issue, the raw vibration signal was first downsampled by 10 (fs=20 kHz) to decrease the input size of each model. The input window size was chosen as 2000 time-domain samples, so that the input to a model includes the motor vibration for at least one revolution of the motor shaft for any shaft speed in this dataset. Also, the samples were augmented by slicing the raw vibration data with 50% overlap (1000 time-domain samples). After this augmentation process, min-max normalization was applied on the input data using the Equation 32.

The 1D Self-ONN model used for bearing fault diagnosis on this dataset has 3 operational and 2 dense layers as shown in Figure 26. The 1D Self-ONN layers has 32, 24, and 16 neurons with kernel sizes 49, 27 and 5 respectively. There are 20 neurons in the hidden MLP layer and the output dense layer size is 5 which is equal to the number of classes. At the output MLP layer, cross-entropy loss with softmax function was used. The hyperbolic tangent activation function *tanh* was utilized through all 1D Self-ONN and MLP layers. The subsampling factors were selected as 16, 8 and 8 respectively through 1D Self-ONN layers. The best test accuracy for each case was encountered for q=1 meaning 1D CNN. For all cases in Table 10, 3 different BP runs were performed. Among each BP runs and epochs, the model with the minimum validation loss was selected as the best performing model. This model was then tested on the test dataset. The same Adam optimizer (lr = 0.001) discussed earlier in the text, was used for this dataset as well and the batch size was 32.

Bearing Health conditions	Speed varying conditions			
	Increasing speed	Decreasing speed	Increasing then decreasing speed	Decreasing then increasing speed
Healthy	H-A-1	H-B-1	H-C-1	H-D-1
	H-A-2	H-B-2	H-C-2	H-D-2
	H-A-3	H-B-3	H-C-3	H-D-3
Faulty (inner race fault)	I-A-1	I-B-1	I-C-1	I-D-1
	I-A-2	I-B-2	I-C-2	I-D-2
	I-A-3	I-B-3	I-C-3	I-D-3
Faulty (outer race fault)	O-A-1	O-B-1	O-C-1	O-D-1
	O-A-2	O-B-2	O-C-2	O-D-2
	O-A-3	O-B-3	O-C-3	O-D-3
Faulty (ball fault)	B-A-1	B-B-1	B-C-1	B-D-1
	B-A-2	B-B-2	B-C-2	B-D-2
	B-A-3	B-B-3	B-C-3	B-D-3
Faulty (combination of faults)	C-A-1	C-B-1	C-C-1	C-D-1
	C-A-2	C-B-2	C-C-2	C-D-2
	C-A-3	C-B-3	C-C-3	C-D-3

Figure 25. Sub-datasets of University of Ottawa's bearing data. (Green: dataset X, yellow: dataset Y, cyan: dataset Z).

Table 10. 6 cases for training, validation and test splits of University of Ottawa bearing data, and the test accuracy of the proposed 1D CNN model.

Case	**Dataset X (3800 samples)**	**Dataset Y (3800 samples)**	**Dataset Z (3800 samples)**	**1D CNN Test Accuracy (%)**
1	Training	Validation	Test	99.7
2	Training	Test	Validation	99.9
3	Validation	Training	Test	94.7
4	Validation	Test	Training	95.1
5	Test	Training	Validation	99.9
6	Test	Validation	Training	99.1
Average Test Accuracy				**98.1**

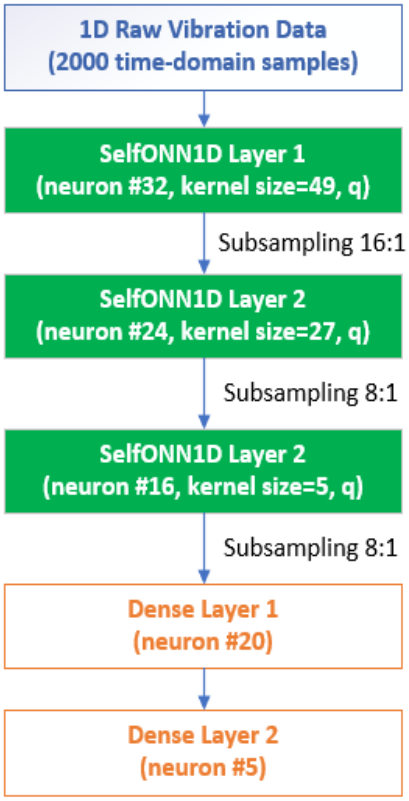

Figure 26. 1D Self-ONN Classifier (for University of Ottawa bearing data) with 1D Self-ONN (number of neurons, kernel size, degree of the Taylor approximation) and dense layers (number of neurons).

The maximum number of epochs was chosen as 20. The test accuracy of the proposed 1D CNN model is given in Table 10. As we can see, when the 1D CNN model sees all the speed varying conditions, its diagnosis accuracy becomes quite satisfactory with the average test accuracy of 98.1%. This model has 24,741 trainable parameters and takes raw vibration data at the input, thus allows real-time implementation.

To compare the performance of the proposed model with the existing deep learning-based methods, another scenario was considered. For this case, the model was trained using the data files in the increasing speed column and tested using the data on

the decreasing speed column of Figure 25. The number of classes is 5, and the same 1D Self-ONN model in Figure 26 (q=1, 3, 5, and 7) was used for this case.

Table 11. Comparison of different methods in terms of classification accuracy on University of Ottawa bearing data. (Training data: increasing speed, Test data: decreasing speed).

Model	**Classification Accuracy (%)**	**Trainable Parameters**
1D CNN	97.5	24,741
1D Self-ONN (Q=3)	90.6	73,189
1D Self-ONN (Q=5)	81.3	121,637
1D Self-ONN (Q=7)	85.4	170,085
(Bera, Dutta and Dhara, 2021)	95.9	419,621
VGG16	97.3	165,738,309
ResNet50V2	96.2	23,575,045
InceptionV3	95.5	21,778,597

30% of increasing speed data was used for validation, and the remaining 70% was used for training. For this case, 1D CNN model with 24,741 trainable parameters has the highest diagnosis accuracy with 97.5% as shown in Table 11. Training, validation, and test accuracies of the proposed 1D CNN model over each training epochs is given in Figure 27. In Table 12, confusion matrix for the proposed 1D CNN architecture is given.

Table 12. Confusion matrix of 1D CNN model on University of Ottawa bearing data. (Training data: increasing speed, Test data: decreasing speed).

		Predicted Label				
		Healthy	**IR**	**OR**	**BF**	**Combined**
True Label	**Healthy**	569	0	0	1	0
	IR	0	570	0	0	0
	OR	0	0	570	0	0
	BF	69	1	0	500	0
	Combined	0	0	0	0	570

(Bera, Dutta and Dhara, 2021) trained a deep 2D CNN model with 6 convolutional layers by inputting the spectrograms images of increasing speed vibration data, and then evaluated their model on the test set consisting of spectrograms of bearings subjected to decreasing speed. They also trained pre-existing DL models such as VGG16, Residual Network (ResNet50V2) and Inception Network (InceptionV3) using the same increasing speed training data and evaluated each model on the decreasing speed test data. Table 11 summarizes diagnosis accuracies of the 1D CNN, Self-ONN, and existing DL-based methods. 1D CNN (q=1) and Self-ONN (q=3,5,7) performance results on University of Ottawa's bearing data (training data: increasing speed, test data: decreasing speed) is also presented in Table 13. The shallow and robust 1D CNN architecture proposed for this dataset demonstrated a superior performance compared to the pre-existing networks.

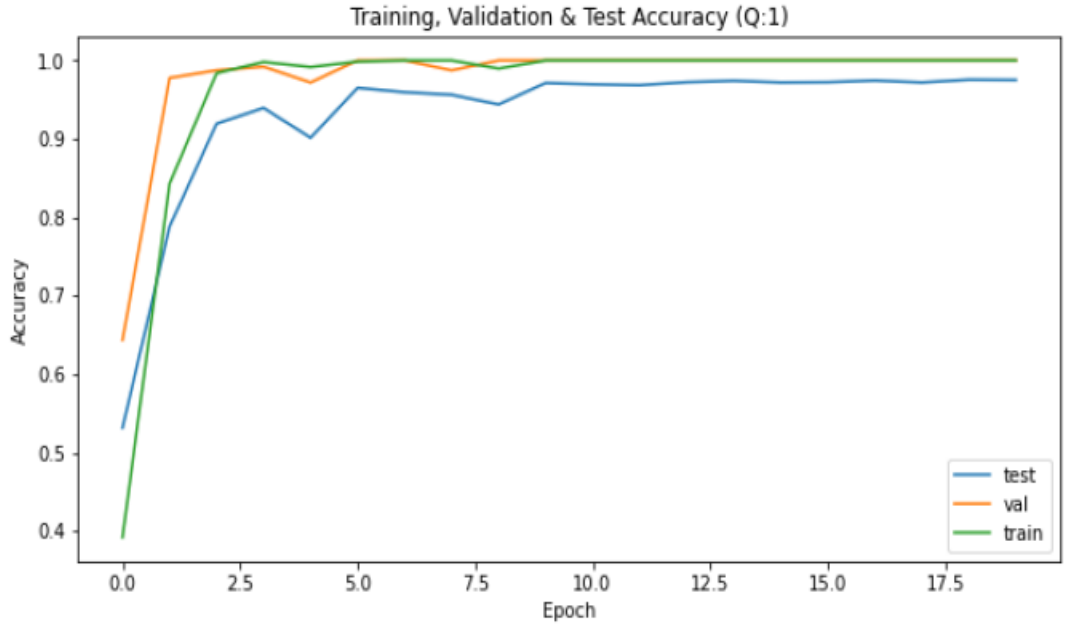

Figure 27. Training, validation, and test accuracies of the proposed 1D CNN model over each training epochs.

Table 13. 1D CNN (q=1) and Self-ONN (q=3,5,7) results for University of Ottawa's bearing dataset. (Training data: increasing speed, Test data: decreasing speed).

Train/Val/Test	**Training = 1994, Validation = 859, Test = 2850 Samples**					
Q	**q=1**			**q=3**		
Scores	**Precision**	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Healthy	0.892	0.998	0.942	0.763	0.944	0.844
IR	0.998	1.000	0.999	0.910	0.996	0.951
OR	1.000	1.000	1.000	0.986	0.998	0.992
BF	0.998	0.877	0.934	0.992	0.649	0.785
Combined	1.000	1.000	1.000	0.942	0.944	0.943
Accuracy			0.975			0.906
Q	**q=5**			**q=7**		
Scores	**Precision**	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Healthy	0.631	0.977	0.767	0.641	0.951	0.766
IR	0.743	0.989	0.849	0.866	0.979	0.919
OR	1.000	0.719	0.837	1.000	0.996	0.998
BF	0.996	0.405	0.576	0.984	0.432	0.600
Combined	0.981	0.974	0.977	0.959	0.912	0.935
Accuracy			0.813			0.854

CHAPTER 5: DEPLOYMENT ON MICROCONTROLLERS

5.1. Quantization of Neural Networks

In an intelligent IoT device that relies on neural networks for decision making, data is generally collected and stored on the cloud with wireless communication technologies such as WiFi and BLE. Then, using the collected data, a neural network is trained on the cloud. When the new data arrive from the device to the cloud, the inference is made on this unseen data and the output prediction is again sent back to the device to make a decision. This is the lifecycle of a cloud computing system, and it comes with several drawbacks. Since the data is transferred wirelessly to a server for further processing, privacy is the first concern especially for the security critical applications in a cloud computing system. Secondly, permanent connectivity should be ensured, otherwise non-deterministic latencies may occur, which results in interruption of the whole system. Furthermore, if the data throughput rate is very high, the device may consume lots of power each time it sends the data to the cloud either for training of the neural network or for the inference. Another alternative to the cloud computing is the edge computing and it resolves some of the problems mentioned above. Edge computing tries to minimize the circulation of the data. By doing so, data is not transmitted to the cloud for inference, thus the inference is made on the device itself. However, neural networks are usually not trained on the IoT devices in these systems, instead the network is often trained offline on a local server placed closer to data. There are also some approaches to both train and run the inference on the device, but in this thesis, these approaches are not discussed. The proposed machine monitoring and diagnostics system works as follows. First, the data is collected wirelessly from the device for a couple of minutes, and then using the collected data, a neural network is trained offline on a workstation. After that, the neural network is quantized and deployed on the IoT device to run inference periodically. At this stage, the device can run standalone, and it will only transfer predictions made on the device to lower power consumption. On the other hand, running neural networks on a resource-constrained device like microcontrollers is a challenging task (Novac *et al.*, 2021). Firstly, NN algorithms run slower on microcontrollers than on GPUs or CPUs, since the clock frequency is much lower (8 MHz to 80 MHz compared to 1GHz to

2GHz), and parallelism techniques such as thread-level parallelism or advanced vectorization are not usually implemented (Novac et al., 2021). Microcontrollers typically include a general-purpose processing core, and require less power compared to their counterparts, thus allowing battery powered devices. However, in a scenario where the device is not accessible after installation, it requires long battery life, so power consumption could be a major problem. Finally, the most major issue may be memory constraint of microcontrollers. These devices usually have very small amount of memory, i.e., often less than 1MB**.**

A technique to shrink the size of a neural network to be deployed on the microcontrollers is called quantization. Quantizing a neural network means reducing the number of bits used to encode each weight and/or activation of a model, while keeping desired accuracy. Using quantization, total memory usage of a network can be reduced by a considerable amount.

5.1.1. Quantization Fundamentals

Modern computing systems use floating-point to represent real numbers, and 32-bit single-precision floating point is the common format used in deep learning frameworks. Although neural network models are usually trained using the 32-bit single-precision format, alternative formats such as IEEE fp16 and bfloat16 have also been implemented recently to reduce training duration and memory usage without significant loss in performance (Burgess et al., 2019). Once the neural network is ready to be deployed on a device after training, it can be quantized using even lower precision formats such as fixed-point and integer. By using lower precision formats during inference, we may accelerate math-intensive operations like convolution and matrix multiplication. Furthermore, memory bandwidth and memory size requirements can be alleviated using lower precision formats, so latency can be reduced with less memory access and simpler computations. As a result, we can observe reduced power consumption for both computations and memory access.

Quantization can be divided into two main categories: non-uniform and uniform. While non-uniform quantization uses non-linear transformations, in uniform quantization, step sizes are equal. An example of weight distribution of a convolutional layer kernel, i.e., conv1d/kernel_0 of the 1DCNN model trained using our dataset, is shown in Figure 28. It can be observed from this figure that when the input is normalized, convolutional layer weight distribution is like a Gaussian distribution with the mean near 0. Therefore, they could be better represented with a non-uniform quantization, and floating-point representation ensures a better precision around 0. In non-uniform quantization, a non-linear function needs to be computed beforehand offline or online to generate a lookup table to get a non-constant quantization step. This brings an additional overhead and can still cause slight quantization error. On the other hand, since our aim is to perform fast computations, uniform quantization with constant quantization step is often preferable.

Quantization to lower precision format could be automated by some integrated software tools such as Tensorflow Lite Micro (TFLM) and STM32Cube.AI. Since these tools usually implement uniform integer quantization (integer weights and activations) for neural network inference, the rest of this section will explain the fundamentals of this approach.

Uniform quantization can be performed in two steps. Firstly, quantization range, i.e., the range of real numbers to be quantized, should be selected, and the values outside of this range are clamped. Then, the real values are mapped to integers according to the number of bits used for the quantized representation. That means each mapped real value is rounded to the closest integer value. The two essential operations named “Quantize” and “Dequantize” are needed to allow integer manipulations in a pre-trained floating-point neural network. We can convert a floating-point (e.g., fp32) number to a quantized integer (e.g., int8) using “Quantize” operation. The dual “Dequantize” operation can then convert a quantized integer (e.g., int32) back into a real number (e.g., fp16). Floating-point formats like fp16 and fp32 are here considered as real numbers for the purpose of discussion.

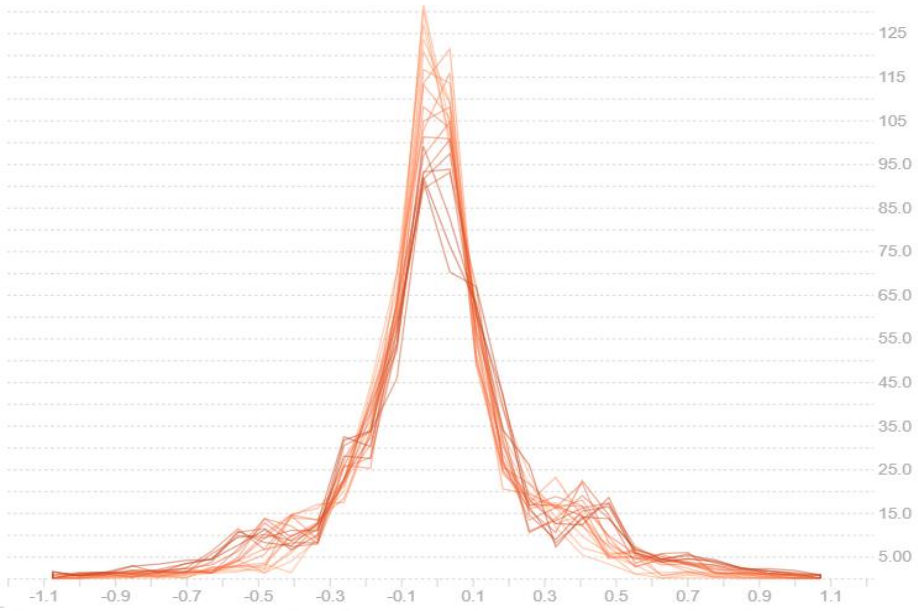

Figure 28. An example of weight distribution for a convolutional layer kernel (conv1d/kernel_0).

We can now define the operations "Quantize" and "Dequantize". Let's define the quantization range, i.e., the range of real numbers to be quantized, as $[q_{min}, q_{max}]$ and the bit-width of the signed or unsigned integer as *n*. A real value *x* in the range $[q_{min}, q_{max}]$ can now be represented as a signed or unsigned integer in the range $[-2^{n-1}, 2^{n-1} - 1]$ and $[0, 2^n - 1]$ respectively. The inputs outside this range are simply clipped. As we are only focusing on the uniform transformation, there are two options. The first one is uniform affine quantization (also known as asymmetric quantization), and the second one, which is a special case of the first, is the symmetric uniform quantization (also called scale quantization).

5.1.1.1. Uniform Affine Quantization

Uniform affine quantization (also called asymmetric quantization) is controlled by the following parameters: the scale factor *s*, the bit-width *n* and the zero-point *z*. The scale factor *s* is a floating-point number which controls the step size. On the other hand, the zero point is an integer, and it must be chosen such that real zero is quantized

without error. When these parameters are defined, we can formulate the quantization operation. A real valued *x* can be mapped into the unsigned integer grid as:

$$\mathrm{x}_{int} = \mathrm{clamp}\left(round\left(\frac{x}{s}\right) + z\,;\ 0, 2^n - 1\right)$$

Equation 34. Asymmetric quantization.

where *round* is round-to-nearest operator, and clamp is defined as:

$$\mathrm{clamp}(f; a, b) = \begin{cases} a, & x < a \\ x, & a \le x \le b \\ b, & x > b \end{cases}$$

Equation 35. Clamp function.

Then, we can approximate the real valued *x* from its integer representation as:

$$\mathrm{x} \cong \hat{x} = s(x_{int} - z)$$

Equation 36. Dequantization step of asymmetric quantization.

If we plug x_{int} in the Equation 34 into Equation 36, we can get the general quantization function q as:

$$\hat{x} = \mathrm{q}(x; s, z, n) = \mathrm{s}\left[clamp\left(round\left(\frac{x}{s}\right) + z\,;\ 0, 2^n - 1\right) - z\right]$$

Equation 37. General quantization function.

From the quantization step in Equation 34, one can determine the quantization range $[q_{min}, q_{max}]$ as:

$$\mathrm{q}_{min} = -sz$$
$$\mathrm{q}_{max} = \mathrm{s}(2^n - 1 - z)$$

Equation 38. Quantization limits q_{min} and q_{max}.

If there are any values outside of this quantization range, they will be clipped to quantization limits introducing a clipping error. To reduce this clipping error, one can expand the quantization range by increasing the scale factor, but this, in turn, brings about increased rounding error (Nagel et al., 2021).

5.1.1.2. Symmetric Uniform Quantization

Symmetric uniform quantization is a subset of uniform affine quantization, where the zero point is located at 0. Real numbers can be quantized either to signed or unsigned integers using symmetric quantization. Unsigned symmetric quantization is often preferred when one-tailed distributions (e.g., ReLU activations) are encountered, while the signed symmetric quantization is mostly used for distributions which are roughly symmetric about zero (e.g., the weight distribution of a convolutional layer kernel).

To map a real valued *x* into a signed integer using symmetric quantization, we can perform the following operation:

$$\mathrm{x}_{int} = \mathrm{clamp}\left(round\left(\frac{x}{s}\right)\ ;\ -2^{n-1}, 2^{n-1} - 1\right)$$

Equation 39. Signed symmetric quantization.

On the other hand, to perform unsigned symmetric quantization, one can use the Equation 40.

$$\mathrm{x}_{int} = \mathrm{clamp}\left(round\left(\frac{x}{s}\right)\ ;\ 0, 2^{n} - 1\right)$$

Equation 40. Unsigned symmetric quantization.

Finally, we can approximate the real valued *x* from its integer representation as:

$$\hat{x} = sx_{int}$$

Equation 41. Dequantization step of symmetric quantization.

Although both symmetric and asymmetric quantization allow us to use integer arithmetic, asymmetric quantization results in more computationally expensive inference due to its extra offset parameter. To show this fact, let us consider the multiplication of asymmetric activations with asymmetric weights as given in Equation 42.

$$\begin{aligned}\widehat{\boldsymbol{W}}\widehat{\boldsymbol{x}} &= s_w(\boldsymbol{W}_{int} - z_w)s_x(\boldsymbol{x}_{int} - z_x) \\ &= s_w s_x \boldsymbol{W}_{int}\boldsymbol{x}_{int} - s_w s_x z_x \boldsymbol{W}_{int} - s_w s_x z_w \boldsymbol{x}_{int} + s_w s_x z_w z_x\end{aligned}$$

Equation 42. Multiplication of asymmetric activations with asymmetric weights.

If both the weights and activations were quantized with symmetric quantization, then we would only get the first term in Equation 42. The second and fourth terms can be pre-computed, since they depend only on the zero-point, scale and weight values which are known in advance. On the other hand, the third term includes the input data *x*, and it requires an additional computation during inference. Hence, this additional computation results in increased latency and power consumption. For this reason, symmetric weight quantization, which avoids the additional input-dependent term, is widely used in most of the integrated software tools like Tensorflow

Lite Micro. For a bit-width of 8, three different examples of uniform quantization are illustrated in Figure 29.

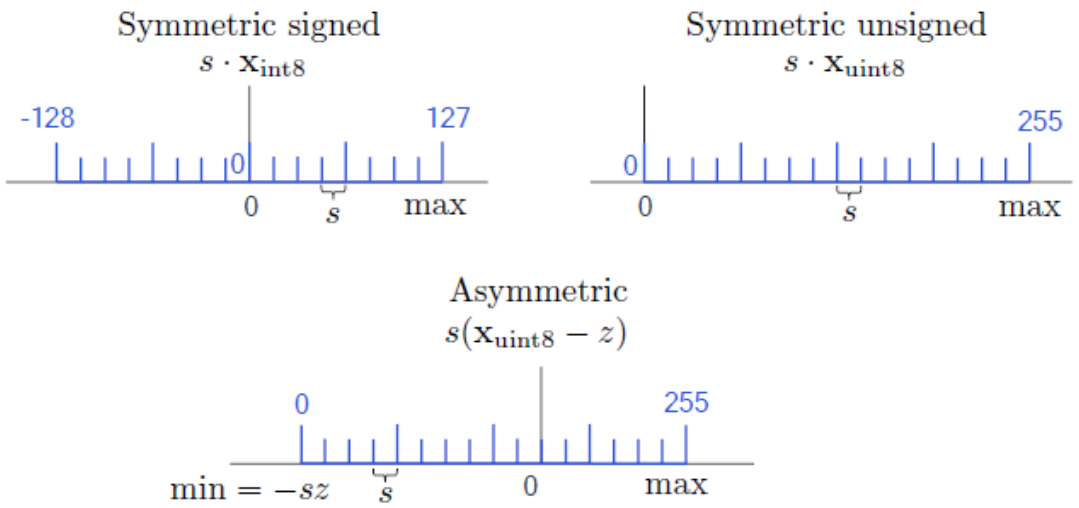

Figure 29. An illustration of symmetric and asymmetric uniform quantization for a bit-width of 8. The floating-point grid is in black, and the integer quantized grid is shown in blue (Source: Nagel et al., 2021).

Quantization parameters can be shared among tensor elements and the way they are shared specifies the quantization granularity. Per-tensor quantization is the most widely used granularity since it allows easy hardware implementation. In per-tensor quantization, all the accumulators in Equation 42 will use the same scale factor, $s_w s_x$. On the other hand, we can achieve a finer granularity with per-channel quantization. If a different quantizer is used per-channel for a 3D tensor (e.g., per-kernel or equivalently per-output-channel), accuracy might improve especially when the distribution varies significantly from channel to channel.

Figure 30 illustrates an overview of how a quantized matrix-vector multiplication is performed in a neural network hardware. The fundamental computation performed during the inference is Multiply–Accumulate operation (MAC). Accumulators are first loaded with the bias value. Then, the weight and input values are loaded into separate arrays, and their product are calculated in the

corresponding processing elements. Finally, their results are added in the accumulators. In this example, int8 arithmetic is used, but any bit-width can be selected for the quantization. To avoid overflow, a higher bit-width accumulators are generally used (e.g., int32). 32-bit accumulators, in the given example, store the activations, and to reduce the data transfer and allow next layer's operations without complexity, these activations are quantized back into int8.

Figure 30. An illustration of MAC operation for quantized inference (Source: Nagel et al., 2021).

5.1.1.3. Quantization Range

To quantize a floating-point tensor, we first need to decide the quantization limits q_{min} and q_{max} for model weights and activations. Weights can be simply quantized without using any calibration data, but variable tensors such as model input, output and activations cannot be calibrated unless a few inference cycles are run. Some calibration methods are min-max, entropy, and percentile.

Min-max calibration method uses the whole dynamic range of the tensor, and the quantization limits q_{min} and q_{max} can be found as follows:

$$q_{min} = \min \boldsymbol{W}$$

$$q_{max} = \max \boldsymbol{W}$$

Equation 43. Min-max calibration.

where $\boldsymbol{W}$ is the tensor to be quantized.

This method does not introduce clipping error, but it is sensitive to outlier which may cause rounding error. In entropy calibration method, Kullback-Leibler divergence is used to minimize the information loss between the original floating-point values and the quantized format. Finally, in percentile method, the range is set to a percentile of the distribution of absolute values seen during calibration. For example, 99% calibration clips 1% of the largest magnitude values.

5.1.2. Post-Training Quantization

Post training quantization is a training-free quantization, and it is the most preferred option by user. In post-training quantization, we first train the neural network model in a floating-point format (e.g., fp32). When the training is done, the model is frozen, and its parameters are quantized. The quantized model is finally deployed on the target to run inference. Quantization of the weights and inputs usually results in a quantization error, which, in turn, introduces a quantization error on the activations. Depending on the bit-width used, this quantization error may accumulate through each layer and end up in a wrong prediction at the output. If the bit-width of parameters decreases, the quantization error usually increases, causing an accuracy drop compared to the original floating-point model. However, in some cases, although the quantization error increases slightly, the neural network can generalize better on the unseen test data.

5.1.3. Quantization-Aware Training

The post-training quantization (PTQ) is very easy and fast to implement since we do not need to retrain the neural network with labeled data. However, when we quantize a real number to a lower precision format like 4-bit integer, we usually encounter a significant accuracy drop (Nagel et al., 2021). This accuracy drop can be minimized using the quantization-aware training. In quantization-aware training, backpropagation steps remain the same, and weights and biases are still stored in floating point format so that they can be updated by small amounts. On the other hand, the low precision behavior is simulated in forward propagation pass to adjust the parameters and minimize the loss introduced by the quantization. Training with simulated quantization is illustrated in Figure 31. First, a training graph of the floating-point model is created. Then, fake quantization nodes are inserted to the locations where tensors are represented by fewer bits during inference. We finally train the model in simulated quantized mode until it converges. Now, the optimized inference graph is ready to be deployed on a target device.

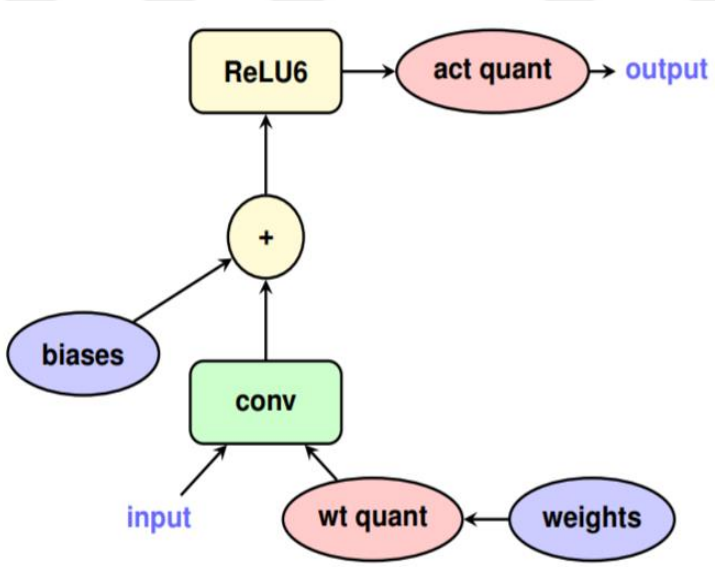

Figure 31. Training with simulated quantization (Quantization-aware Training).

5.2. Embedded AI Frameworks

Embedded artificial intelligence (AI) enables inexpensive and low-power AIoT solutions and several embedded AI frameworks such as STM32Cube.AI and TensorFlow Lite for Microcontrollers (TFLM), accelerates the path to successful real-world deployment of ML models. In this section, the most popular AI frameworks STM32Cube.AI and TFLM, will be discussed in detail.

5.2.1. STM32Cube.AI

STM32Cube.AI from STMicroelectronics is an integrated software tool that enables the generation of STM32-optimized library from a pre-trained neural network. It accelerates the deployment of some neural network models like MLPs and CNNs into STM32 microcontrollers. STM32Cube.AI supports numerous machine learning frameworks including TensorFlow Lite and Keras, and the models in ONNX format can also be implemented.

STM32Cube.AI can run inference using both floating-point (if the target has FPU) and fixed-point (8-bit integers) formats. TensorFlow Lite quantized models and 8-bit quantization of Keras networks are both supported.

STM32Cube.AI ecosystem is fully integrated with the other STMicroelectronics development tools; thus, it allows easy and fast implementation. The tool itself has several ways to validate ML models both on the computer and STM32, and the performance on STM32 devices can be measured without user handmade ad hoc C code (STMicroelectronics, 2021).

To optimize the inference performance, STM32Cube.AI utilizes Common Microcontroller Software Interface Standard Neural Network (CMSIS-NN) library. CMSIS-NN is a collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint of neural networks on Cortex-M processors (Arm, 2021). The library has separate functions to operate on different

weight and activation data types including 8-bit and 16-bit integers. The inference engine uses a proprietary library, so it is not allowed to manipulate it.

5.2.2. TensorFlow Lite for Microcontrollers

TensorFlow Lite for Microcontrollers (TFLM) is an open-source embedded ML framework originally derived from TensorFlow Lite. It is primarily designed to run ML models on microcontrollers with a few kilobytes of memory. TFLM is written in C++ 11 and requires a 32-bit platform. Hence, the inference code is portable. It may also generate the inference code for development environments like Mbed. Various neural network architectures such as MLPs, CNNs and RNNs can be deployed on the target 32-bit microcontroller using TFLM.

To deploy and run TensorFlow models on a 32-bit microcontroller, one should perform the following steps. Firstly, a TensorFlow model with supported operations is trained. Then, this model is converted to a Tensorflow Lite model using TensorFlow Lite converter Python API. This will convert the model into a FlatBuffer, reducing the model size, and modify it to use TensorFlow Lite operations (TensorFlow Lite, 2021). At this stage, we could consider using post-training quantization so that we can obtain a smaller model size. Finally, the model is converted to a C byte array (C source file that contains the TensorFlow Lite model as a char array) using standard tools to be included in the program. Using this source file and TFLM C++ library, we can compile the program and run inference on the target device.

TFLM can also run inference using both floating-point (if the target has FPU) and fixed-point (8-bit integers) formats. Both PTQ and QAT methods are supported in TFLM. While 8-bit symmetric quantization is used for weights, activations are quantized using asymmetric quantization in TFLM. 32-bit integers are used for biases to avoid high accuracy drop. Convolution operation can be implement using per-filter scale factor and offset, but other operations usually use per-tensor (i.e., per-layer) scale factor and offset. The 8-bit integer inference performance can again be optimized using CMSIS-NN library kernels. These kernels utilize Single Instruction Multiple Data

(SIMD) instructions. The SIMD instructions enable simultaneous calculations for two 16-bit operations or four 8-bit operations, so they greatly enhance the inference performance on ARM microcontrollers.

TFLM provides several post-training quantization options such as dynamic range quantization, full-integer quantization, and float-16 quantization. The decision tree given in Figure 32 can help us decide which PTQ method is best for the use case.

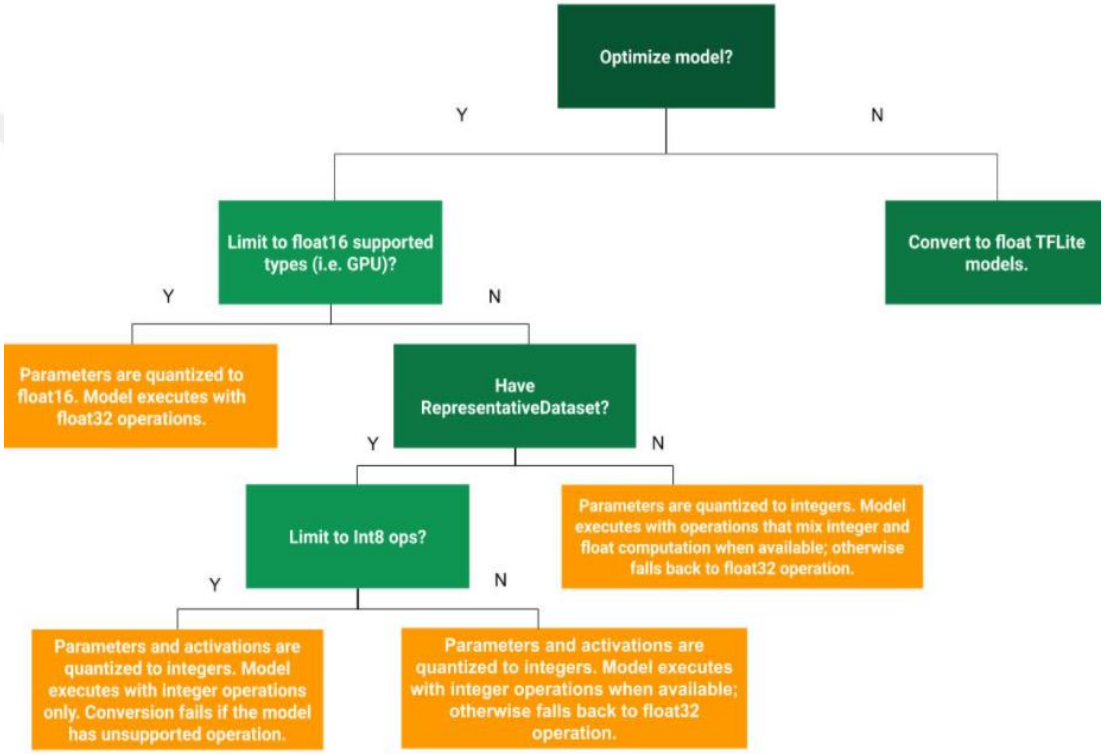

Figure 32. TFLM PTQ decision tree (Source: TensorFlow Lite, 2021).

CHAPTER 6: EXPERIMENTAL SETUP AND ON-DEVICE PERFORMANCE RESULTS

6.1. Platform Description

An experimental motor test platform was set up in electrical machines laboratory at Izmir University of Economics. As shown in Figure 34, the test stand includes a single-phase induction motor (right), a variac (not shown) to adjust the voltage, and the B-L475E-IOT01A2 Discovery kit (inside 3D printed case) for data acquisition and ML inference on the edge. 5 single-phase induction motors were used. Three of them were VM 90S-2 220V, 1.5 kW and 2880 rpm single-phase induction motors, and the condition of the bearings for these electric motors were healthy, outer-race and inner-race fault. The remaining two were VM 90S-4 220V, 1.1 kW and 1400 rpm single-phase induction motors and the health condition of bearings for these motors were healthy and ball fault. All the motors were from Volt Electric Motors company. Single-point faults were introduced to the drive-end bearings with a fault diameter of 1.5 mm. Ball bearings with fan-end bearing type 6203 ZZ and drive-end bearing type 6205 ZZ were employed in all electric motors used in this setup. Sample fault introduced and healthy bearings are shown in Figure 33. The B-L475E-IOT01A2 Discovery kit was securely fastened to the electric motor cooling fin edges using an aluminum mounting bracket as shown in Figure 35.

Figure 33. Sample healthy and fault introduced ball bearings.

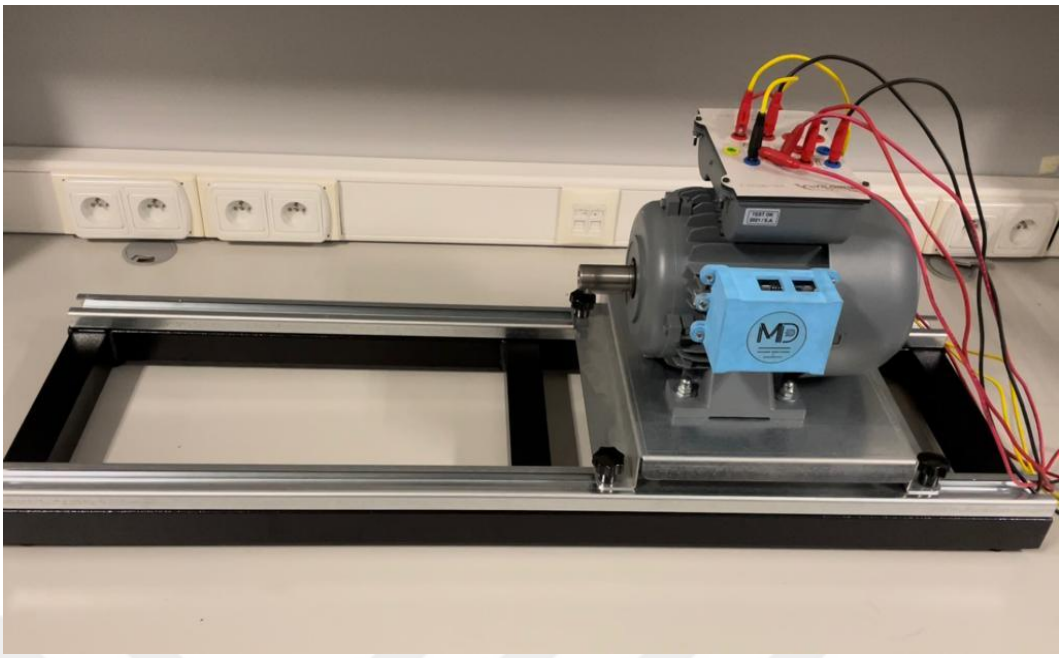

Figure 34. Machine monitoring and diagnostics (MMD) test stand.

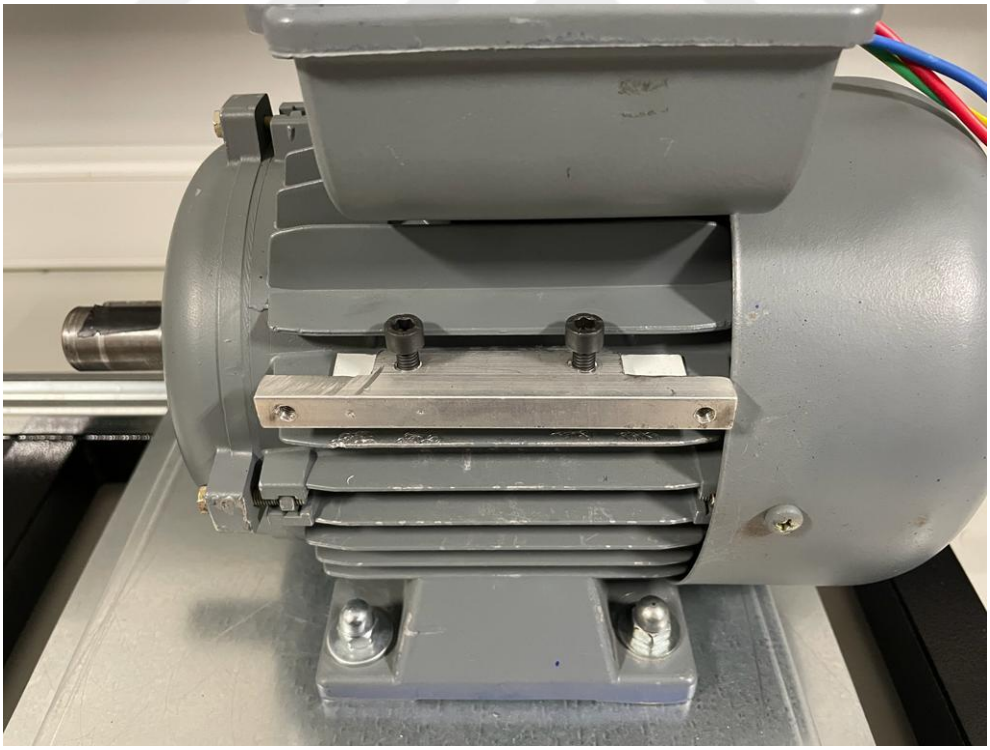

Figure 35. Aluminum mounting bracket.

6.2. Development Board - STM32L4 Discovery Kit IoT Node

The B-L475E-IOT01A Discovery kit was used for motor acceleration data collection, and on-device ML inference. The Discovery kit includes ultra-low-power STM32L4 Series MCU based on Arm Cortex-M4 core with 1 Mbyte of Flash memory and 128 Kbytes of SRAM. Several STMicroelectronics sensors are available on this board such as 3D accelerometer and 3D gyroscope (LSM6DSL), 3-axis magnetometer (LIS3MDL), capacitive digital sensor for relative humidity and temperature (HTS221), and 2 on-board omnidirectional digital microphones (MP34DT01). The board also includes a wide range of on-board communication modules such as Bluetooth (V4.1), Wi-Fi, sub-GHz RF module, and a dynamic NFC tag. The layout of this board is shown in Figure 36.

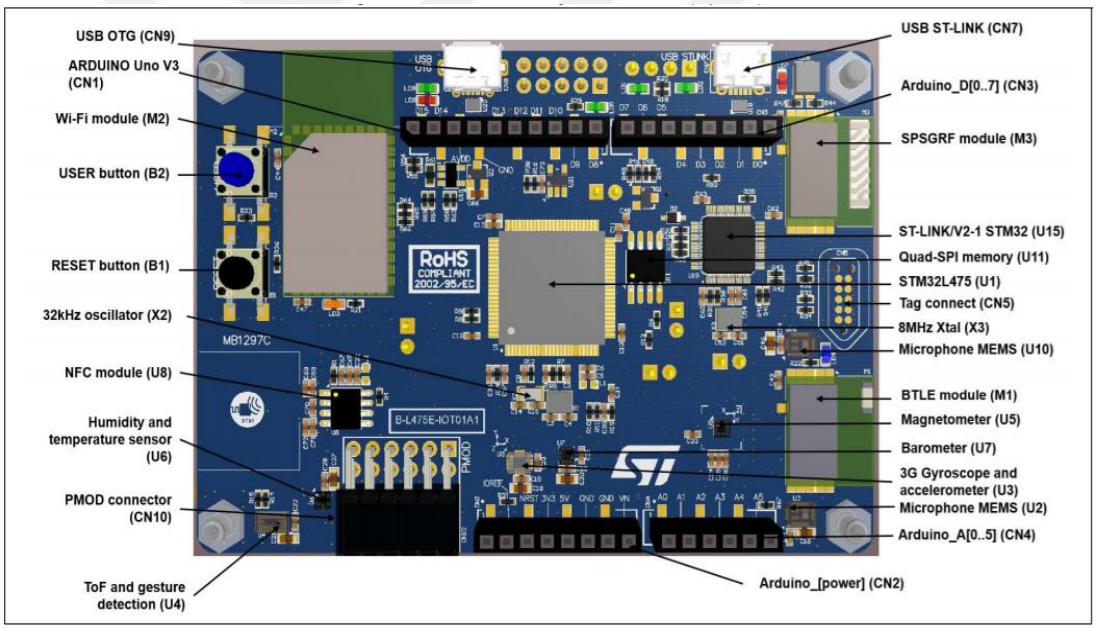

Figure 36. STM32L4 Discovery Kit IoT Node.

The main reason behind using this board was its low-power 3D accelerometer and 3D gyroscope (LSM6DSL). 3-axis (XYZ) raw acceleration data from this on-board accelerometer was used to diagnose bearing faults of single-phase induction motors. The full-scale acceleration range of LSM6DSL is user-specified, and the available values are ±2/±4/±8/±16 g. It offers output data rates (ODRs) from 12.5 Hz up to 6.66 kHz in high-performance mode.

This board is Arm Mbed enabled, thus it allows rapid prototyping using its open-source IoT operating system. It offers a well-defined API to write C++ applications with free tools, libraries, and drivers for common components. Mbed OS was also used in this thesis for the embedded machine learning application development.

6.3. On-Device Performance Results

The overall system block diagram is shown in Figure 37. The proposed intelligent machine monitoring and diagnostics system works as follows. First, the data can be collected from the device for a couple of minutes over the USB serial port or wirelessly using the on-board Wi-Fi or BLE module. Then, using the collected data, a neural network model can be trained offline on a workstation or on a cloud platform. After that, the neural network is trained and quantized, and the corresponding ML files are exported. At this stage, the new binary can be compiled and deployed on the IoT device to run inference periodically. From then on, the device can run standalone, and it may only transfer predictions made on the device to lower power consumption.

To demonstrate the working of this intelligent system, 3-axis raw acceleration data was collected from the mentioned single-phase induction motors on the motor test stand. The output data rate (ODR) of the on-board accelerometer was set to its maximum value 6.66 kHz, and the full-scale acceleration range was selected as ±4 g. This choice of full-scale range was made by observation of the raw acceleration waveform to avoid saturation. A simple Mbed program that reads 3-axis raw acceleration with these settings was compiled and run on the board. This program periodically reads 1 second long (6660 time-domain samples) 3-axis raw acceleration and send these samples over the serial port. These samples can then be printed on a serial monitor using a serial monitor application like Tera Term. By this way, for each healthy state and fault condition, several minutes of 3-axis raw acceleration data was collected under no load condition. The length of collected data for each case is given in Table 14.

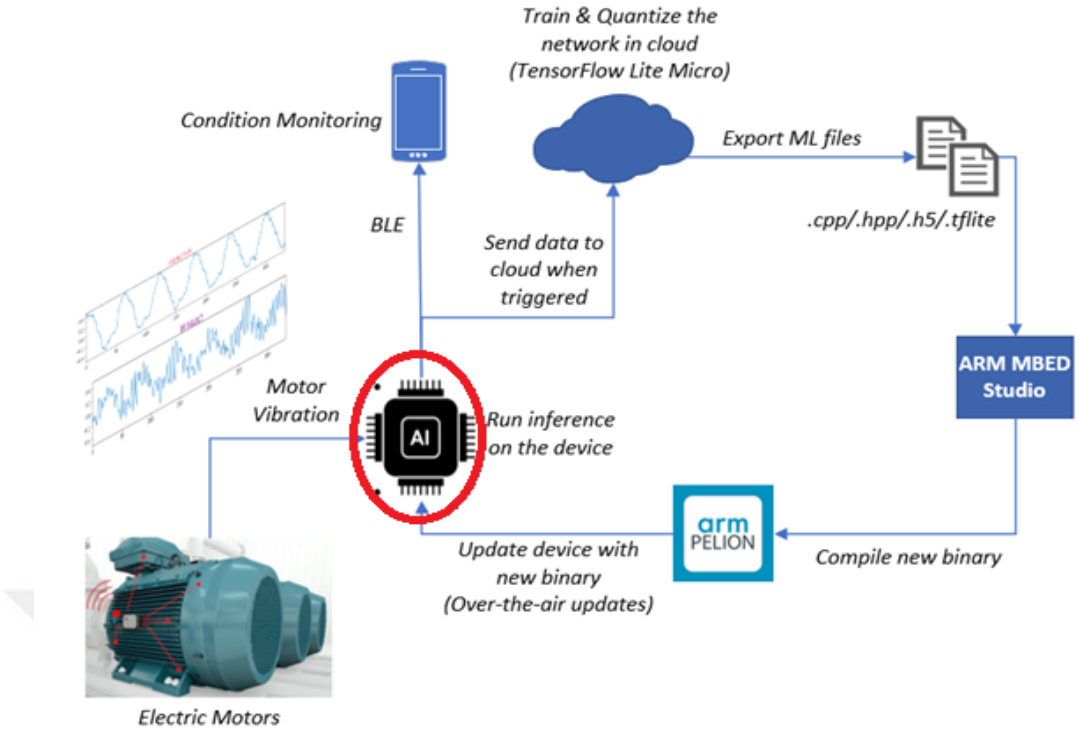

Figure 37. Microcontroller based motor fault detection and diagnosis system.

Table 14. MMD bearing fault dataset

Motor (Bearing) Health Condition	Motor Load (hp)	Motor Type	Length of Dataset (minutes)
Idle	-	-	4
Healthy	0	VM 90S-4 (1400 rpm)	9
Healthy	0	VM 90S-2 (2880 rpm)	9
Inner Race Fault	0	VM 90S-2 (2880 rpm)	9
Outer Race Fault	0	VM 90S-2 (2880 rpm)	9
Ball Fault	0	VM 90S-4 (1400 rpm)	9

Sample faulty and healthy motor raw acceleration waveforms and the amplitude spectrum of the z-axis acceleration are shown in Figure 38 through Figure 40.

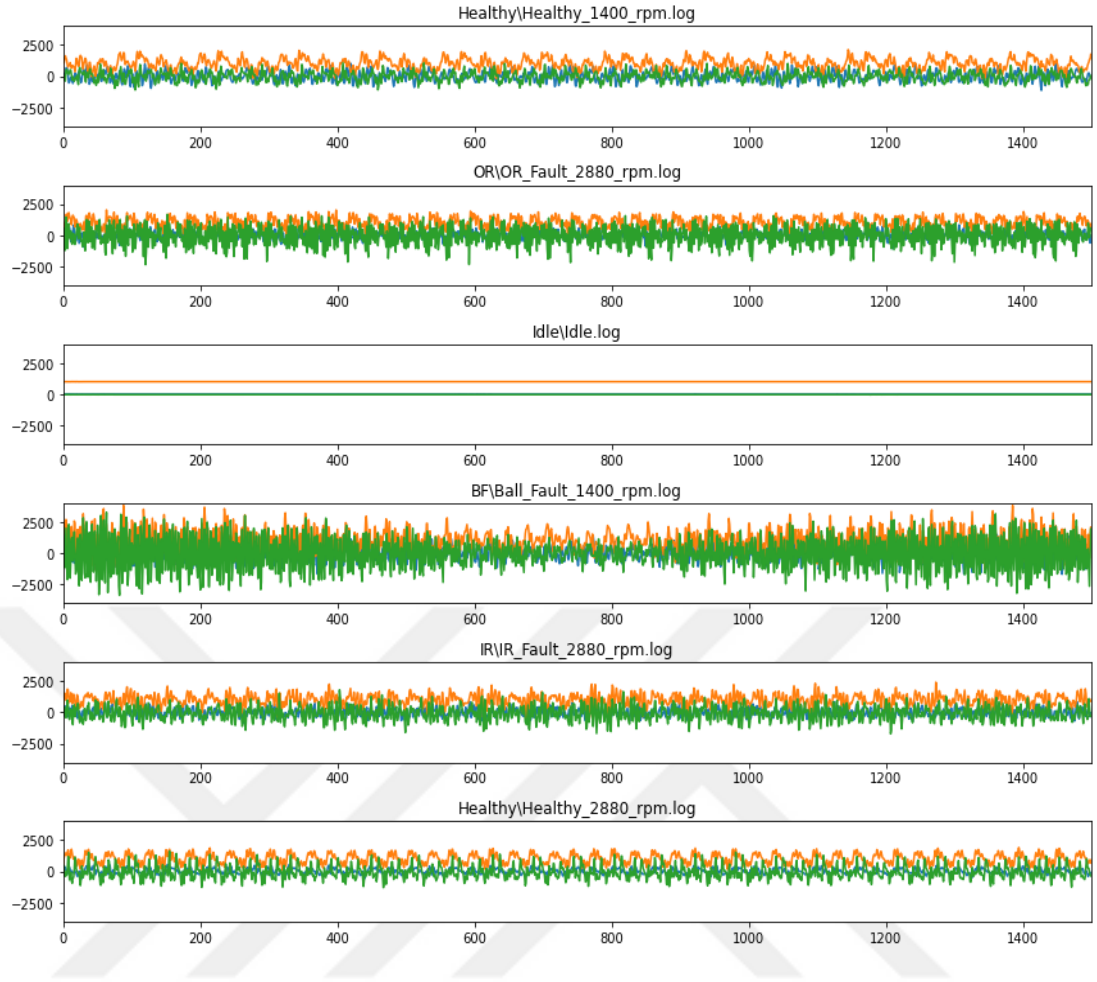

Figure 38. 1500 samples healthy and faulty motor 3-axis raw acceleration waveforms. (X axis: blue, Y axis: orange, Z axis: green)

After the data collection process, a 2D CNN model was trained using the collected data. The input to this model was 3-axis raw acceleration. The input window size was chosen as 333 time-domain samples so that the raw acceleration includes at least one revolution of the motor shaft. The whole data was divided into 4 equal pieces, and each of them has the examples from all classes. Two of them (50%) was used for training, one (25%) for validation and one (25%) for testing. Therefore, there are 29,400 training, 14,700 validation and 14,700 test samples in total. After the segmentation process, mean normalization was also applied on the input data to subtract the mean from each channel using the Equation 44.

$$x_{scaled} = \frac{x - mean(x)}{max(x) - min(x)}$$

Equation 44. Mean normalization.

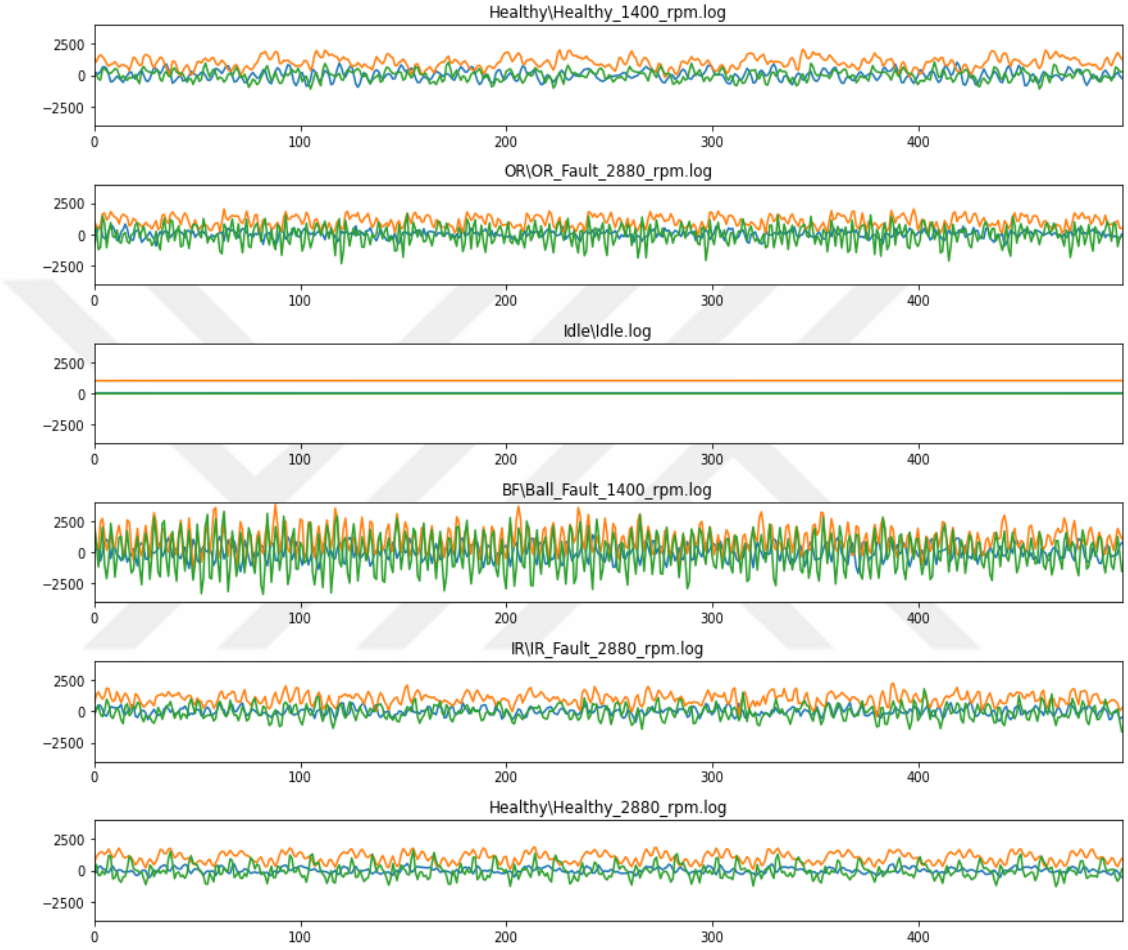

Figure 39. 500 samples healthy and faulty motor 3-axis raw acceleration waveforms. (X axis: blue, Y axis: orange, Z axis: green).

Since both Tensorflow Lite Micro (TFLM) and X-CUBE-AI expansion package support 1D convolution by adding a singleton dimension to 2D convolution, a 2D CNN was used for deployment. The 2D CNN model used for this dataset is shown in Figure 41. It has 3 convolutional and 2 dense layers. Normalized 3-axis raw acceleration data with size 333 (time-domain samples) was inputted to the model. The 2D convolutional layers has 16, 8, and 8 neurons with kernel sizes 11, 7 and 3

respectively. There are 10 neurons in the hidden MLP layer, and the output dense layer size is 5 which is equal to the number of classes. These 5 classes are idle, healthy, outer-race fault, inner-race fault, and ball fault. Idle state is used to indicate that the electric motor is not running. At the output MLP layer, cross-entropy loss with SoftMax function was used. The rectified linear activation function "ReLU*"* was utilized through all convolutional and MLP layers except the output. ReLU activation is simpler to implement on a microcontroller, since it outputs the input directly if it is positive, otherwise, it will output zero. The Adam optimizer with a learning rate of 0.001 was used. The default batch size of 32 was used. The number of epochs was 20. Early stopping was utilized by monitoring validation loss with patience 5 epochs. The training was carried out on a TensorFlow version 2.4.1 installed PC with 2.00GHz Intel Core i7-4510U CPU and NVIDIA GeForce 840M graphic card.

To reduce the feature map size through the convolutional layers, convolution with stride 4 was used instead of max-pooling for the following reasons. Stride is an argument that controls the step size as a convolution filter is slid across the input. Many neural network frameworks use the stride size 1 as the default value. However, if we set the stride to 2, each window will be offset by two samples from its neighbor, and this results in an output array that is half the width and height of the input. As a result, the output of each convolution layer occupies much less memory, resulting in reduced RAM usage, and the computation is also reduced with increased stride size that allows fast inference. Larger stride values may cause some accuracy loss that can be verified during training. However, as it reduces the MCU resource usage dramatically, one can increase the other parameters like the number of neurons and gain some accuracy back.

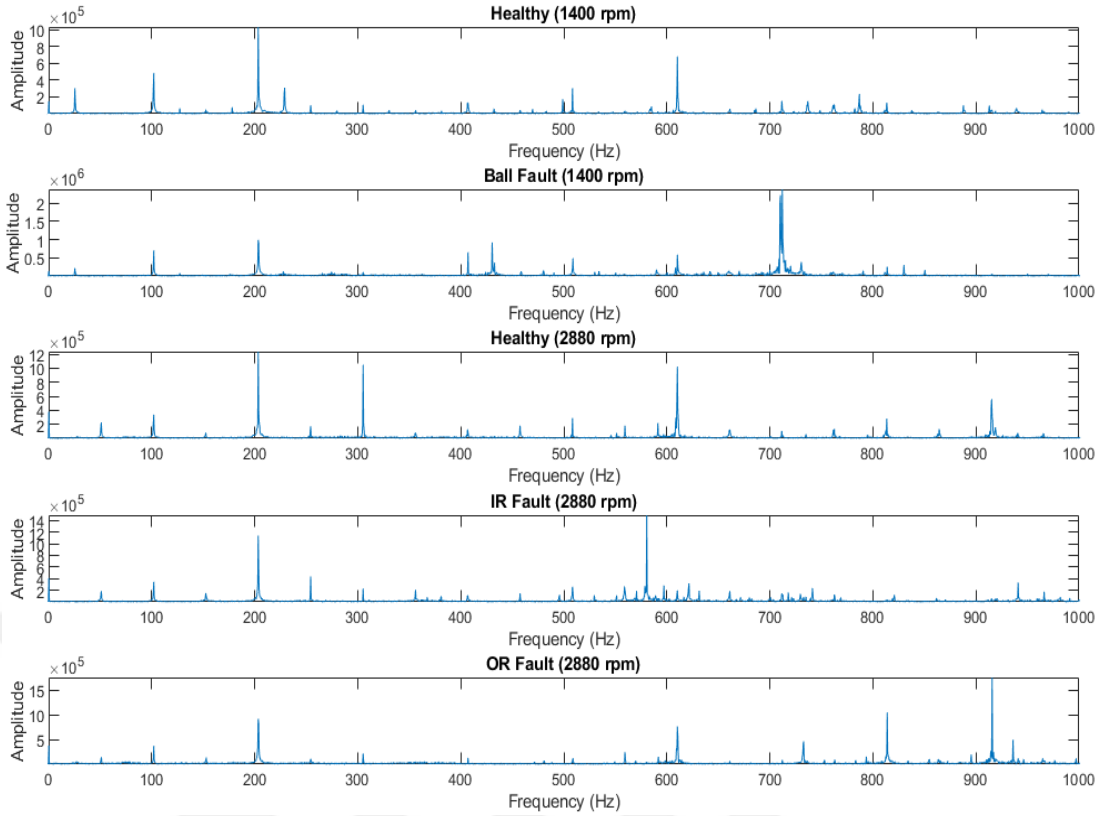

Figure 40. The amplitude spectrum of z-axis acceleration.

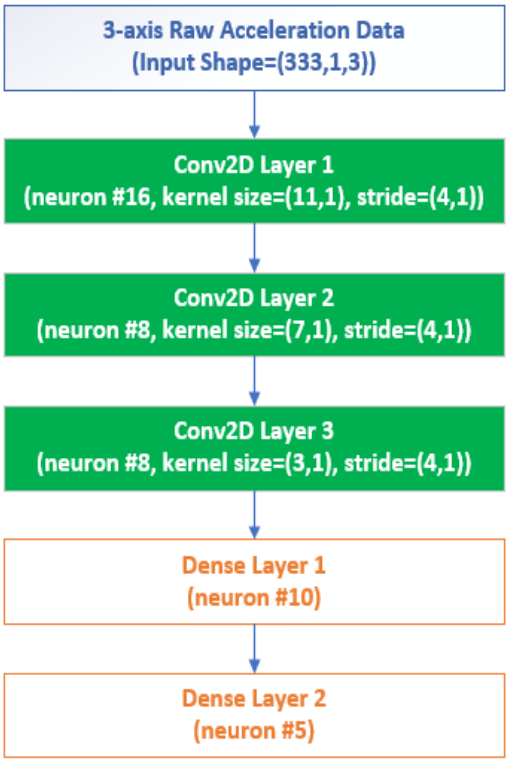

Figure 41. 2D CNN model with 3 conv2D and 2 dense layers.

After training the model, we need to follow some steps to deploy and run a TensorFlow model on a microcontroller using the TFLM run-time. First, the TensorFlow model should be converted to a TensorFlow Lite model using the TensorFlow Lite converter Python API. This step converts the model into a FlatBuffer format, reducing the model size, and modify it to use TensorFlow Lite operations. At this stage, to get a smaller model size, one can use post-training quantization (PTQ) or the model can be trained using quantization-aware training (QAT). The TensorFlow Lite model is then converted to a C byte array using standard tools to store it in read-only program memory on the microcontroller. Using this C byte array and the TFLM C++ library, we can run inference on the microcontroller. One can also use STM32Cube.AI ecosystem to deploy a TensorFlow Lite model on microcontrollers. As a part of the STM32Cube.AI ecosystem, X-CUBE-AI is an expansion package that enables automatic conversion and integration of pre-trained neural networks. An optimized library is generated automatically and included in the user's project.

For comparison, several 2D CNN models were deployed on the STM32L4 Discovery Kit IoT Node using both TFLM and STM32Cube.AI runtime. In terms of model size, the advantages of strided convolution in CNNs compared to max-pooling can be observed by comparing the same models in the Table 15 and Table 16. For the same number of neurons and filter size in convolutional layers, using stride of 4 instead of max-pooling with pool size of 4, RAM usage can be significantly reduced with no significant reduction in test accuracy. The other benefit is the fast inference which can be seen from Table 17 and 18. In these tables, the average duration for 10 inferences is reported. The inference duration can be reduced to a few milliseconds using strided convolution which enables real-time implementation. The other important decision to make is the quantization type. In PTQ, as the number of parameters in the model is reduced, the test accuracy of the quantized model drops considerably for smaller networks. To avoid such accuracy loss, we can utilize quantization-aware training.

Table 15. The comparison of float and quantized model size and test accuracy for strided convolution (strides=4) using TFLM and STM32Cube.AI runtimes.

Model 3 Conv. + 2 FC {Neurons} (Kernel Size) [Stride]	Number of Trainable Parameters	Quantization Type	Model Test Accuracy		Model Size (Bytes)			
			Float	Quantized	Float		Quantized	
					Flash	RAM	Flash	RAM
{16}(11)[4], {8}(7)[4], {8}(3)[4], {10}, {5}	2,113	PTQ Full-Integer (Int 8)	0.99919	0.99907	11728	11104	7448	4288
{8}(11)[4], {4}(7)[4], {4}(3)[4], {10}, {5}	817	PTQ Full-Integer (Int 8)	0.99890	0.88662	6544	8384	5592	3520
{8}(11)[4], {4}(7)[4], {4}(3)[4], {10}, {5}	817	QAT (Int 8)	0.99890	0.99919	6544	8384	6280	7024

Table 16. The comparison of float and quantized model size and test accuracy for max-pooling (pool size=4) following convolutional layers using TFLM and STM32Cube.AI runtimes.

Model 3 Conv. + 2 FC {Neurons} (Kernel Size)	Number of parameters	Quantization Type	Model Test Accuracy		Model Size (Bytes)			
			Float	Quantized	Float		Quantized	
					Flash	RAM	Flash	RAM
{16}(11), {8}(7), {8}(3), {10}, {5}	2,033	PTQ Full-Integer (Int 8)	1.0	1.0	12004	28208	8152	8928
{8}(11), {4}(7), {4}(3), {10}, {5}	777	PTQ Full-Integer (Int 8)	1.0	0.99931	6988	16624	6336	5952
{8}(11), {4}(7), {4}(3), {10}, {5}	777	QAT (Int 8)	1.0	0.99971	6988	16624	6888	7536

Table 17. The comparison of float and quantized model test accuracy and inference speed for max-pooling (pool size=4) following convolutional layers using TFLM and STM32Cube.AI runtimes.

Model 3 Conv. + 2 FC {Neurons} (Kernel Size)	Number of parameters	Quantization Type	Model Test Accuracy		TFLM Run-time Inference Duration (msec)		STM32Cube.AI Run-time Inference Duration (msec)	
			Float	Quantized	Float	Quantized	Float	Quantized
{16}(11), {8}(7), {8}(3), {10}, {5}	2,033	PTQ Full-Integer (Int 8)	1.0	1.0	123.441	25.318	64.495	38.605
{8}(11), {4}(7), {4}(3), {10}, {5}	777	PTQ Full-Integer (Int 8)	1.0	0.99931	58.366	16.197	31.278	24.660
{8}(11), {4}(7), {4}(3), {10}, {5}	777	QAT (Int 8)	1.0	0.99971	58.366	16.916	31.278	25.448

Table 18. The comparison of float and quantized model test accuracy, and inference speed for strided convolution (strides=4) using TFLM and STM32Cube.AI runtimes.

Model 3 Conv + 2 FC {Neurons} (Kernel Size) [Stride]	Number of Trainable Parameters	Quantization Type	Model Test Accuracy		TFLM Run-time Inference Duration (msec)		STM32Cube.AI Run-time Inference Duration (msec)	
			Float	Quantized	Float	Quantized	Float	Quantized
{16}(11)[4], {8}(7)[4], {8}(3)[4], {10}, {5}	2,113	PTQ Full-Integer (Int 8)	0.99919	0.99907	30.079	6.340	15.617	3.873
{8}(11)[4], {4}(7)[4], {4}(3)[4], {10}, {5}	817	PTQ Full-Integer (Int 8)	0.99890	0.88662	14.202	4.059	7.543	2.647
{8}(11)[4], {4}(7)[4], {4}(3)[4], {10}, {5}	817	QAT (Int 8)	0.99890	0.99919	14.202	4.758	7.543	3.354

All the quantized models given on the previous tables can fit on the target MCU, since it has 128 Kbytes of SRAM and 1 Mbyte of Flash memory. The first model in Table 18 was used for deployment. To visualize the 3-axis raw acceleration and the prediction of the proposed CNN on a phone, Phyphox BLE Mbed library was used. In contrast to classic Bluetooth, BLE is designed for significantly lower power consumption, thus the BLE devices can run for weeks or months on a coin cell battery. This encourages us to communicate with the development board using the on-board Bluetooth V4.1 module (SPBTLE-RF). The board was powered up by a 3.7V Li-Po battery with an external boost converter to step up the voltage to 9V. Phyphox application was used on mobile. A custom BLE service with two characteristics were created. The first characteristic was used to send 3-axis acceleration data, and the second one was used to send the predictions made by the ML model. The compiled source code first reads 3-axis acceleration from the on-board accelerometer sensor and applies mean normalization to each axis. Then, the normalized data is inputted to the ML model and TensorFlow's interpreter-based inference engine runs the inference. The 3-axis acceleration data and the prediction made by the model can be sent to a GATT client by notifications. Notifications are unacknowledged; thus, they are faster. This process can be called periodically to reduce power consumption. The experiment created in the Phyphox application for bearing fault diagnosis is given in Figure 42. Finally, a video demonstration of this system can be watched from the following link (https://ieucloud.izmirekonomi.edu.tr/index.php/s/fEDAXNzDsFnkfJO).

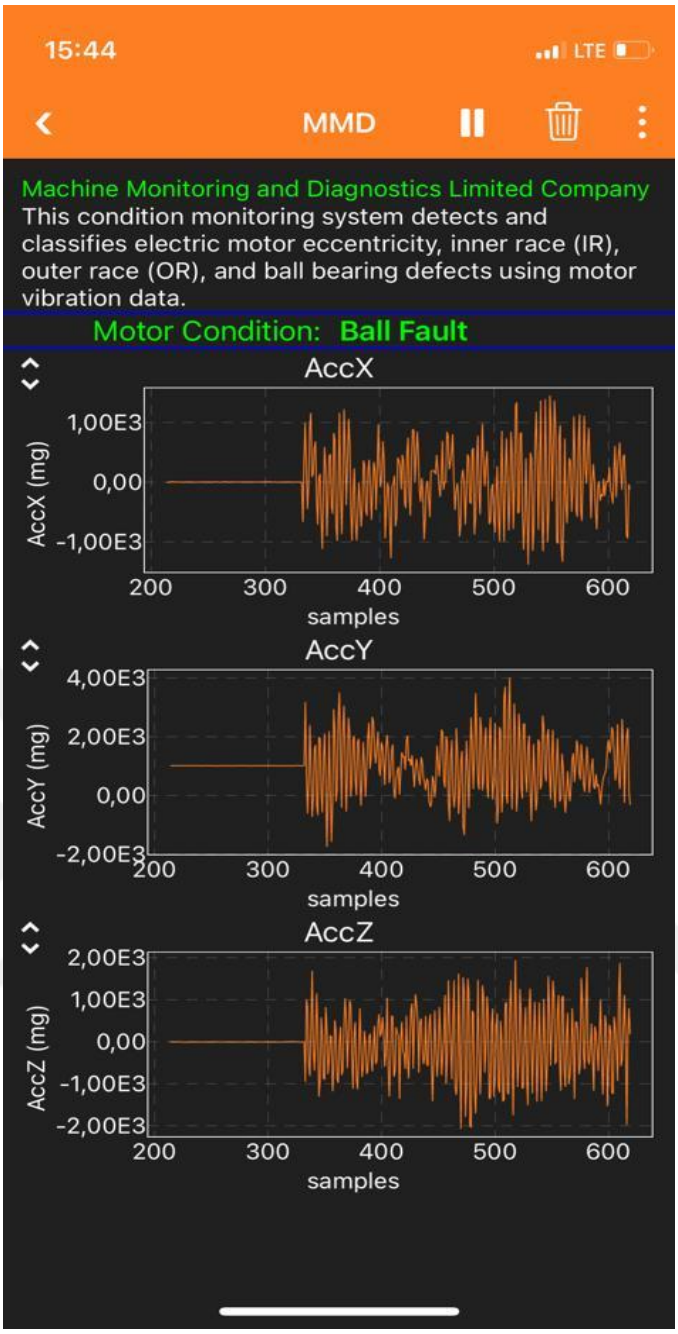

Figure 42. The designed Phyphox experiment that shows bearing health condition and the 3-axis motor vibration on a mobile.

CHAPTER 7: CONCLUSION

Continuous machine monitoring is of great importance and obviously a crucial challenge in the industry. Unexpected machine failures and breakdowns result in costly replacement of machine components and unplanned production downtime. REBs are commonly used in rotating machinery, and they are one of the most major causes of induction machine faults. For this reason, rolling element bearing fault diagnosis techniques has been studied extensively for decades.

The earlier signs of rolling element bearing failure can most easily be detected using an accelerometer, and it can be used as an effective predictive maintenance tool to estimate when the maintenance should be performed. In a highly digitized and connected production facility, with the increasing number of sensors used in the field, access to high volume of data has now become possible. Therefore, the use of deep learning techniques for BFDD has increased day by day.

Deep learning-based bearing fault diagnosis methods have been discussed in detail throughout this thesis. Two open-source benchmark datasets, i.e., Case Western Reserve University (CWRU) and University of Ottawa bearing vibration data, were used for this purpose. In domain and cross domain (under different loading conditions) performances of 1D CNNs and Self-ONNs for bearing fault diagnosis were evaluated using the CWRU bearing data. In addition, the performance of 1D CNN and Self-ONNs was evaluated under time-varying rotational speed conditions using University of Ottawa bearing data. This work was the first in terms of cross domain evaluation of 1D CNNs and Self-ONNs for bearing fault diagnosis, and they were also tested under time-varying rotational speed conditions. From the results, the superior performance of 1D Self-ONNs over 1D CNNs are obvious for cross domain tests, demonstrating better generalization capability at more difficult tasks. However, when the data is not so varied and the number of samples are sufficient for each class, the performance of 1D CNNs is quite satisfactory.

To further evaluate the performance of 1D CNNs, from two different single-phase induction motors with four different bearing health conditions (healthy, outer-race, inner-race, and ball fault), real motor acceleration data was collected using the on-board 3-axis accelerometer of STM32L4 Discovery kit IoT node. A 1D CNN model was then trained, quantized, and deployed to the microcontroller using the collected data, and the embedded AI tools TensorFlow Lite for Microcontrollers (TFLM) and STM32Cube.AI. These tools were compared in terms of average inference time and memory footprint for this application. Post training quantization (PTQ) and quantization-aware training (QAT) were also discussed and the effect of quantization in the test accuracy was observed. The experimental results show that 1D CNNs can easily be deployed on microcontrollers and used for real-time bearing fault diagnosis. A system with BLE connectivity was employed for continuous monitoring of bearing health condition of single-phase induction motors. As a future work, the main target is to deploy 1D Self-ONNs onto the microcontrollers to utilize their generalization capability across different load domains. Furthermore, the data collection process will be accelerated through cloud connectivity rather than just using the BLE connection.

REFERENCES

Arm. (2021). Common Microcontroller Software Interface Standard (CMSIS) [Online]. Available at: https://developer.arm.com/tools-and-software/embedded/cmsis. (Accessed: 15 December 2021).

Bera, A., Dutta, A. and Dhara, A. K. (2021) *Deep learning based fault classification algorithm for roller bearings using time-frequency localized features*, Proceedings - IEEE 2021 International Conference on Computing, Communication, and Intelligent Systems, ICCCIS 2021, pp. 419–424.

Bloch, H. P., and Geitner, F. K. (1999) *Machinery Component Failure Analysis*, Practical Machinery Management for Process Plants, Vol. 2, pp. 79–256.

Burgess, N., Milanovic, J., Stephens, N., Monachopoulos, K., and Mansell, D. (2019) *Bfloat16 Processing for Neural Networks*, IEEE 26th Symposium on Computer Arithmetic, pp. 88–91.

Case Western Reserve University (CWRU). (2004) Bearing Data Center [Online]. Available at: https://engineering.case.edu/bearingdatacenter (Accessed: 17 July 2021).

Chen, X., Zhang, B., and Gao, D. (2021) *Bearing fault diagnosis base on multi-scale CNN and LSTM model*, Journal of Intelligent Manufacturing, Vol. 32, pp. 971–987.

Cheng, C., Zhou, B., Ma, G., Wu, D., and Yuan, Y. (2020) *Wasserstein distance based deep adversarial transfer learning for intelligent fault diagnosis with unlabeled or insufficient labeled data*, Neurocomputing, Vol. 409, pp. 35–45.

Ding, X. and He, Q. (2017) *Energy-Fluctuated Multiscale Feature Learning With Deep ConvNet for Intelligent Spindle Bearing Fault Diagnosis*, IEEE Transactions on Instrumentation and Measurement, Vol. 66, pp. 1926–1935.

Du, W., Tao, J., Li, Y., and Liu, C. (2014) *Wavelet leaders multifractal features based fault diagnosis of rotating mechanism*, Mechanical Systems and Signal Processing, Vol. 43, pp. 57–75.

Eren, L. (2017) *Bearing fault detection by one-dimensional convolutional neural networks*, Mathematical Problems in Engineering, Vol. 2017, pp. 1-9.

Eren, L., Ince, T. and Kiranyaz, S. (2019) *A Generic Intelligent Bearing Fault Diagnosis System Using Compact Adaptive 1D CNN Classifier*, Journal of Signal Processing Systems, Vol. 91, pp. 179–189.

Ergin, S., Uzuntas, A. and Gulmezoglu, M. B. (2012) *Detection of stator, bearing and rotor faults in induction motors*, Procedia Engineering, Vol. 30, pp. 1103–1109.

Ferreira, F. J. T. E. and De Almeida, A. T. (2012) *Induction motor downsizing as a low-cost strategy to save energy*, Journal of Cleaner Production, Vol. 24, pp. 117–131.

Ganin, Y., Ustinova, E., Ajakan, H., Germain, P., Larochelle, H., Laviolette, F., Marchand, M., and Lempitsky, V. (2016) *Domain-Adversarial Training of Neural Networks*, Journal of Machine Learning Research, vol. 17, pp. 1–35.

Gao, S., Wang, X., Miao, X., Su, C., and Li, Y. (2019) *ASM1D-GAN: An Intelligent Fault Diagnosis Method Based on Assembled 1D Convolutional Neural Network and Generative Adversarial Networks*, Journal of Signal Processing Systems, Vol. 10, pp. 1237–1247.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014) *Generative Adversarial Networks*, Communications of the ACM, Vol. 63, pp. 139–144.

Guo, L., Lei, Y., Xing, S., Yan, T., and Li, N. (2019) *Deep Convolutional Transfer Learning Network: A New Method for Intelligent Fault Diagnosis of Machines with Unlabeled Data*, IEEE Transactions on Industrial Electronics, Vol. 66, pp. 7316–7325.

Guo, X., Chen, L. and Shen, C. (2016) *Hierarchical adaptive deep convolution neural network and its application to bearing fault diagnosis*, Measurement, Vol. 93, pp. 490–502.

Harris, T. A. (2001) *Rolling bearing analysis*. 4th Edition. New York: John Wiley & Sons.

Hoang, D. T. and Kang, H. J. (2019) *Rolling element bearing fault diagnosis using convolutional neural network and vibration image*, Cognitive Systems Research, Vol. 53, pp. 42–50.

Huang, H. and Baddour, N. (2019) *Bearing Vibration Data under Time-varying Rotational Speed Conditions*, Data in Brief, Vol. 21, pp. 1745-1749.

Industry Report. (2021). Electric Motor Market Size, Share and Growth [Online] Available at: https://www.fortunebusinessinsights.com/industry-reports/electric-motor-market-100752 (Accessed: 10 January 2021).

Ince, T, Kiranyaz, S., Eren, L., Askar, M., and Gabbouj, M. (2016) *Real-Time Motor Fault Detection by 1-D Convolutional Neural Networks*, IEEE Transactions on Industrial Electronics, Vol. 63, pp. 7067–7075.

Ince, T., Malik, J., Devecioglu, O. C., Kiranyaz, S., Avci, O., Eren, L., and Gabbouj M. (2021) *Early Bearing Fault Diagnosis of Rotating Machinery by 1D Self-Organized Operational Neural Networks*, IEEE Access, Vol. 9, pp. 139260–139270.

Jang, G. B. and Cho, S. B. (2021) *Feature Space Transformation for Fault Diagnosis of Rotating Machinery under Different Working Conditions*, Sensors, Vol. 21, pp. 1417.

Jauregui Correa, J. C. A. and Lozano Guzman, A. A. (2020) *Mechanical Vibrations and Condition Monitoring*. 1st Edition. Academic Press.

Jia, F., Lei, Y., Lin, J., Zhou, X., and Lu, N. (2016) *Deep neural networks: A promising tool for fault characteristic mining and intelligent diagnosis of rotating machinery with massive data*, Mechanical Systems and Signal Processing, Vol. 72-73, pp. 303–315.

Jiang, H., Li, X. Shao, H., and Zhao, K. (2018) *Intelligent fault diagnosis of rolling bearings using an improved deep recurrent neural network*, Measurement Science and Technology, Vol. 29, pp. 065107.

Jiang, W., Hong, Y., Zhou, B., He, X., and Cheng, C. (2019) *A GAN-Based Anomaly Detection Approach for Imbalanced Industrial Time Series*, IEEE Access, Vol. 7, pp. 143608-143619

Jin, X., Zhao, M., Chow, T. W. S. and Pecht, M. (2014) *Motor bearing fault diagnosis using trace ratio linear discriminant analysis*, IEEE Transactions on Industrial Electronics, Vol. 61, pp. 2441–2451.

Kingma, D. P. and Ba, J. L. (2015) *Adam: A method for stochastic optimization*, 3rd International Conference on Learning Representations, pp. 1–15.

Kiranyaz, S., Ince, T., Iosifidis, A., and Gabbouj, M. (2020) *Operational neural networks*, Neural Computing and Applications, Vol. 32, pp. 6645–6668.

Kiranyaz, S., Avci, O., Abdeljaber, O., Ince, T., Gabbouj, M., and Inman D. J. (2021) *1D convolutional neural networks and applications: A survey*, Mechanical Systems and Signal Processing, Vol. 151, p. 107398.

Kiranyaz, S., Malik, J., Abdallah, H. B., Ince, T., Iosifidis, A., and Gabbouj, M. (2021) *Self-organized Operational Neural Networks with Generative Neurons*, Neural Networks, Vol. 140, pp. 294–308.

Kiranyaz, S., Ince, T. and Gabbouj, M. (2016) *Real-Time Patient-Specific ECG Classification by 1-D Convolutional Neural Networks*, IEEE Transactions on Biomedical Engineering, Vol. 63, pp. 664–675.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989) *Backpropagation applied to handwritten zip code recognition*, Neural Computation, Vol. 1, pp. 541–551.

Li, Q., Chen, L., Shen, C., Yang, B., and Zhu, Z. (2019) *Enhanced generative adversarial networks for fault diagnosis of rotating machinery with imbalanced data*, Measurement Science and Technology, Vol. 30, pp. 115005.

Li, Y., Wang, N., Shi, J., Hou, X., and Liu, J. (2018) *Adaptive Batch Normalization for practical domain adaptation*, Pattern Recognition, Vol. 80, pp. 109–117.

Liu, H., Zhou, J., Zheng, Y., Jiang, W., and Zhang, Y. (2018) *Fault diagnosis of rolling bearings with recurrent neural network-based autoencoders*, ISA transactions, Vol. 77, pp. 167–178.

Lu, C., Wang, Z., Qin, W., and Ma, J. (2017) *Fault diagnosis of rotary machinery components using a stacked denoising autoencoder-based health state identification*, Signal Processing, Vol. 130, pp. 377–388.

Lu, W., Wang, X., Yang, C., and Zhang, T. (2015) *A novel feature extraction method using deep neural network for rolling bearing fault diagnosis*, Proceedings of the 2015 27th Chinese Control and Decision Conference, pp. 2427–2431.

Mao, W., Liu, Y., Ding, L., and Li, Y. (2019) *Imbalanced fault diagnosis of rolling bearing based on generative adversarial network: A comparative study*, IEEE Access, Vol. 7, pp. 9515–9530.

Nagel, M., Fournarakis, M., Amjad, R. A., Bondarenko, Y., Baalen, M., and Blankevoort, T. (2021) *A White Paper on Neural Network Quantization*, arXiv preprint

Novac, P. E., Boukli Hacene, G., Pegatoquet, A., Miramond, B., and Gripon, V. (2021) *Quantization and deployment of deep neural networks on microcontrollers*, Sensors, Vol. 21, pp. 2984

Pan, H., He, X., Tang, S., and Meng, F. (2018) *An improved bearing fault diagnosis method using one-dimensional CNN and LSTM*, Journal of Mechanical Engineering, vol. 64, pp. 443–452.

Qian, N. (1999) *On the momentum term in gradient descent learning algorithms*, Neural Networks, Vol. 12, pp. 145–151.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (2013) *Learning Internal Representations by Error Propagation*, Readings in Cognitive Science: A Perspective from Psychology and Artificial Intelligence, Vol. 1., pp. 399–421.

Shao, H., Jiang, H., Zhao, H., and Wang, F. (2017) *A novel deep autoencoder feature learning method for rotating machinery fault diagnosis*, Mechanical Systems and Signal Processing, Vol. 95, pp. 187–204.

STMicroelectronics. (2021). X-CUBE-AI - AI expansion pack for STM32CubeMX [Online]. Available at: https://www.st.com/en/embedded-software/x-cube-ai.html. (Accessed: 10 October 2021).

Tang, H. and Jia, K. (2019) *Discriminative Adversarial Domain Adaptation*, AAAI 2020 - 34th AAAI Conference on Artificial Intelligence, pp. 5940–5947.

TensorFlow Lite. (2021). Build and convert models [Online]. Available at: https://www.tensorflow.org/lite/microcontrollers/build_convert?hl=en. (Accessed: 15 September 2021).

TensorFlow Lite. (2021). Post-training quantization [Online]. Available at: https://www.tensorflow.org/lite/performance/post_training_quantization?hl=en. (Accessed: 11 September 2021).

Wang, F., Dun, B., Deng, G., Li, H., and Han, Q. (2018) *A deep neural network based on kernel function and auto-encoder for bearing fault diagnosis*, I2MTC 2018 - 2018 IEEE International Instrumentation and Measurement Technology Conference: Discovering New Horizons in Instrumentation and Measurement Proceedings, pp. 1–6.

Wang, H., Xu, J., Yan, R., Sun, C., and Chen, X. (2020) *Intelligent Bearing Fault Diagnosis Using Multi-Head Attention-Based CNN*, Procedia Manufacturing, Vol. 49, pp. 112–118.

Xia, M., Li, T., Xu, L., Liu, L., and de Silva, C. W. (2018) *Fault Diagnosis for Rotating Machinery Using Multiple Sensors and Convolutional Neural Networks*, IEEE/ASME Transactions on Mechatronics, Vol. 23, pp. 101–110.

ZHANG, J., Sun, Y., Guo, L., Gao, H., Hong, X., and Song, H. (2020) *A new bearing fault diagnosis method based on modified convolutional neural networks*, Chinese Journal of Aeronautics, Vol. 33, pp. 439–447.

Zhang, W., Peng, G., Li, C., Chen, Y., and Zhang, Z. (2017) *A New Deep Learning Model for Fault Diagnosis with Good Anti-Noise and Domain Adaptation Ability on Raw Vibration Signals*, Sensors, Vol. 17, pp. 425.

Zhang, W., Peng, G. and Li, C. (2017) *Bearings Fault Diagnosis Based on Convolutional Neural Networks with 2-D Representation of Vibration Signals as Input*, MATEC Web of Conferences, Vol. 95, pp. 13001.

Zhao, D., Liu, F. and Meng, H. (2019) *Bearing Fault Diagnosis Based on the Switchable Normalization SSGAN with 1-D Representation of Vibration Signals as Input*, Sensors, Vol. 19, pp. 2000.

Zhou, F., Yang, S., Fujita H., Chen, D., and Wen, C. (2020) *Deep learning fault diagnosis method based on global optimization GAN for unbalanced data*, Knowledge-Based Systems, Vol. 187, pp. 104837.

Zhuang, Z., Lv, H., Xu, J., Huang, Z., and Qin, W. (2019) *A Deep Learning Method for Bearing Fault Diagnosis through Stacked Residual Dilated Convolutions*, Applied Sciences, Vol. 9, pp. 1823.