GENERALIZATION AND AUTOMATION OF MACHINE LEARNING-BASED INTELLIGENT FAULT CLASSIFICATION FOR ROTATING MACHINERY

Justin Larocque-Villiers

A thesis submitted in partial fulfillment of the requirements for the

MASTER OF APPLIED SCIENCE IN MECHANICAL ENGINEERING

Department of Mechanical Engineering
Faculty of Engineering
University of Ottawa

Abstract

This thesis leverages vibration-based unsupervised learning and deep transfer learning to reduce the manual labour involved in building algorithms that perform intelligent fault detection (IFD) on roller element bearings. A review of theory and literature in the field of IFD is presented, and challenges are discussed. An issue is then introduced; current machine learning models built for IFD show strong performance on a small subset of specific data, but do not generalize to a broader range of applications. Signal processing, machine learning, and transfer learning concepts are then explained and discussed. Time-frequency fingerprinting, as well as feature engineering, is used in conjunction with principal component analysis (PCA) to prepare vibration signals to be clustered by a gaussian mixture model (GMM). This process allows for the intelligent referral of data towards algorithms that have performed well on similar datasets and favours the re-use of domain-specific tasks. An algorithm is then proposed that promotes generalization in convolutional neural networks (CNNs) and simplifies the hyperparameter tuning process to allow machine learning models to be applied to a broader set of problems. The machine learning process is then automated as much as possible through meta learning and ensemble models: data similarity measurements are used to evaluate the data fit for transfer and propose training guidelines. Throughout the thesis, three open-source bearing fault datasets are used to test and validate the hypotheses. This thesis focuses on developing and adapting current deep learning models to succeed in challenging domains and real-world scenarios, while improving performance with unsupervised learning and transfer learning.

Acknowledgments

To my family, my friends, and all of whom I have good relationships with, as I truly believe our relationships are what make us human. I wish to thank my wife for her love, support, and patience with me. I wish to thank my sister for her unrelenting care, my mom for the same, and my father for the things he taught me while growing up.

I wish to also thank my supervisor Dr. David Knox for facilitating my introduction to the wonderful yet challenging world of electronics and computer science. And finally, I wish to give a special thank you to my supervisor Dr. Patrick Dumond for being insightful, reliable, and a true team player, even though sometimes I, myself, may have not.

Table of Contents

List of Figures

List of Tables

Nomenclature

ANN: Artificial Neural Network

CNN: Convolutional Neural Network

CBM: Condition Based Maintenance

DNN: Deep Neural Network

CWRU: Case Western Reserve University

DFT: Discrete Fourier Transform

FFT: Fast Fourier Transform

FTFR: First Time Fix Rate

IFD: Intelligent Fault Detection

ML: Machine Learning

MFCC: Mel Frequency Cepstrum Coefficients

MFPT: Machine Failure Prevention Technology

MTTF: Mean Time to Failure

EaaS: Equipment-as-a-Service

PBU: Paderborn University

PdM: Predictive Maintenance

PCA: Principal Component Analysis

REB: Roller Element Bearing

RMSE: Root Mean Square Error

RUL: Remaining Useful Life

SVD: Singular Value Decomposition

Chapter 1

Introduction

1.1 Background

The interconnectedness of machinery with sensors has allowed for optimization of machinery usage through minimization and avoidance of downtime caused by unforeseen breakdowns. Sensors and data processing provide the opportunity to intelligently identify an early-stage fault within a machine prior to a breakdown event. This results in intelligent maintenance strategies revolving around the state of health of the machine, called condition-based monitoring and predictive maintenance (PdM) [1].

When machinery is in operation, the state of health of the subset of components (bearings, gears, pistons, etc.) naturally degrade over time and need replacing. The auxiliary support proponents (lubricant, oil, coolant, etc.) used to support the operation of the machinery also need regular attention [2]. Different types of equipment degrade at different rates. Therefore, keeping manual records of individual degradation is an exhaustive and error-prone task. Ignoring, or otherwise not monitoring the subtle signs of degradation can quickly lead to catastrophes. Equipment failure, depending on the component, can range from small stoppages to severe breakdowns with significant damage or even loss of life. The healthy operation of sub-components, such as bearings and gears, enable the operation of major dependant components, and failure of such major components can lead to sever consequences,

often resulting in the entire equipment assembly breaking down, causing downtime. The cost of downtime varies by industry, with prominent industries being manufacturing, oil and gas, mining, and power generation. For example, in the manufacturing industry, machines tend to be linked in a chain of processes throughout the production line. Therefore, a breakage in one component could cause production wide stoppages, resulting in high costs of downtime. For example, the downtime value in the automotive manufacturing industry may cost up to $22,000 per minute, with potential peaks as high as $50,000 per minute [3].

By intelligently detecting a fault within the machinery prior to a breakdown event, a repair or replacement can be scheduled and optimized for machinery uptime. One of the most common sub-components in rotating machinery are roller element bearings (REBs), which account for a large portion of machinery breakdowns. Rai et al. [4] have assessed that bearings account for between 45%-55% of rotating equipment failures by referencing numerous previous studies on the same topic [5], [6]. Nonetheless, these bearing failures have symptoms that occur as a precursor to a breakdown - with the earliest sign being changes in vibration. This provides an opportunity to detect and prevent the breakdown from occurring. Since the early 1980s, computational methods have been developed and used to analyze machine vibrations to predict whether a fault is present and classify which fault could be the cause [7]. Early methods, which can still be effective today, involve signal processing techniques and time-frequency analysis. These include spectrum analysis, wavelet analysis, cepstrum analysis, and statistical analysis [8], [9].

As computational power became more readily available, principal component analysis (PCA) and support vector machines (SVM) were some of the first machine learning (ML) techniques to be introduced, with many more algorithms emerging throughout the years [10]. The growth of larger datasets and abundance of incoming data through the connectivity of machines further fuelled the growth of ML applications. With more high-quality data

collected (both on healthy operating equipment and on breakdown events), the quality of ML model predictions has improved. Larger datasets allow for more information capture and analysis between the healthy state and failure states of the machine. Specific statistical features can be extracted from the data by engineers and fed to an algorithm for training. K-nearest-neighbours (KNN), random forest (RF), k-means, as well as SVM and PCA are common techniques that rely on a distance measure to classify data into one of many categories or clusters. This distance measure is the principal mechanism that allows for the improvement in performance over signal processing-based techniques. However, in the case of vibration analysis, there tends to be large amounts of data collected (sampling rates of over 10,000 samples per second are common). This results in an abundance of intricate patterns within the data – both in time and frequency – that may not always be linearly related to one another [11]. These intricate patterns and relationships may not be captured by the features used in the feature engineering process due to their dependence on the practicing engineer. Therefore, much of the value and predictability of the signal could be lost. This could also affect performance in noisy environments, which could lead to false negatives or false positives.

In recent work, deep learning has emerged as a viable strategy to circumvent these shortcomings by training deeper models consisting of larger networks of neurons. Deep learning allows for further improvement in classification accuracy and robustness over traditional machine learning techniques by taking into account non-linear behaviour during the classification process [12]. By virtue of the universal approximation theorem, a single-layer network of a sufficient number of neurons can approximate any equation [13]. Although unrealistic, this statement expresses the value of the non-linear nature of deep networks and shifts the importance towards network architecture and data engineering, rather than signal processing and feature engineering. Deep learning can automatically register these hidden

correlations and meanings in the raw data, allowing for more information ingestion and simpler implementation.

The use of deep learning in the predictive maintenance regime is a turning point and a key factor in enabling wide-scale adoption of predictive maintenance programs. However, these new methods have specific challenges. First, rotating machinery exists in a real-world environment, with each application having its own setup; equipment type and size, processes, loads and speeds, noise, etc. Individual variability fuels the requirement for a model that can generalize and perform predictions on a wide range of machinery applications. Second, Faults in the real-world are undesirable and costly, which makes collecting fault signatures rare and scarce. Training new models on machines with unseen faults remains a challenge and fuels the need for existing models that can achieve a good benchmark of performance on a new unseen machine. The performance of deep learning algorithms truly relies on the quality of the data collected. Therefore, these two challenges have significant impact on the future development of this method. Because there is considerably more research fixated on achieving the highest possible accuracy, research on how to properly design algorithms given these constraints is limited and fragmented.

1.2 Thesis Motivations and Goals

PdM strategies benefit greatly from IFD, with rotating machines being particularity influenced by REB research. Increasing the performance of these prediction algorithms results in more feasible and successful programs, which in turn results in cost and efficiency savings. To reduce the barriers to implementation and promote widespread adoption, IFD must address the challenges involved in the implementation of computational models on new and unseen equipment, and not just accuracy improvements on specific train/test splits.

Current research is still focused on achieving state of the art accuracies and high performance on a small range of tasks, which does not address the key challenges stated in this thesis so far, with key solution areas such as knowledge transfer and generalizability remaining under-researched yet highly anticipated. This thesis aims to overcome two specific challenges in the design and implementation of predictive maintenance models:

1. Application-specific overfitting: excessive fine tuning and exhaustive training iterations for improving algorithm performance on specific data distributions, rather than most data distributions.
2. Failure-data scarcity: lack of failure signatures on a given machine specimen.

In addition to the outlined challenges, there are considerations and assumptions that must be made.

- Machinery will have different modes of operation.
- Machinery operation modes will be assumed to be constant throughout the length of a given sample.
- Sensor mounting angles, multiple sensor setups, and multiple data sources, are not considered.

This thesis provides research and understanding into the mechanisms that cause overfitting in the case of IFD using vibration signals from REBs, or more specifically, the mechanisms one can use to avoid overfitting and promote generalization of deep learning. Although high level overfitting is discussed in ML literature, there are a minimal number of strategies put forth for dealing with such challenges in the context of IFD for REBs, for which overfitting is a major concern. The underlying principles discovered are used to propose an architecture that favours generalization between data discrepancies in data domains, and allows for more consistent performance benchmarks, while achieving a reasonable

comparison with existing state of the art. The architecture is tested on commonly used datasets to yield a performance within +/-15% of comparable research benchmarks, while demonstrating a reduction in the complexity of the algorithm, as well as the process used to design it.

The next component of the process, and perhaps the key to enhancing the ML pipeline altogether for IFD on REBs, is the measurement and evaluation of data similarity of the signals coming from different bearing health states. This process allows one to quantify the similarity, or discrepancy, between one data sample and another, and helps adjust the training of an algorithm accordingly. For small data distances, similar architectures from neighbouring models can be pulled and used to quickly achieve reliable results, whereas larger data differences guide the algorithm to adapt the architecture to one more suitable for the given task.

Finally, the first two sections can be used to leverage a generalization favouring architecture, in combination with distance measures to automatically guide the training of algorithms, as well as their selection through an ensemble model. This automation of the machine learning pipeline reduces the time and effort required in designing, developing, and testing a model. The focus is shifted towards data manipulation and guided adjustments to the models. This results in algorithms that are comparable to benchmark performances, without the need for excessive fine tuning. The retraining and fine tuning of algorithms often require a great deal of costly time for engineers, and with the very large number of possible hyperparameter combinations, often leads to random guessing as opposed to educated decision making. By abstracting some of the hyperparameter decision making, the focus of the user can shift towards feeding the right data to the system and making higher level system architecture decisions.

The result of this study is a framework that can guide the decision-making process for a predictive maintenance ML-based strategy adapted to the equipment, type of faults, operating modes and/or conditions, amount of data, and other variables provided. It abstracts many of the decisions from the user, allowing the user to focus on value building activities rather than iterative retraining.

1.3 Thesis Organization and Contributions

Chapter 2 conducts a broad literature review of maintenance strategies, and more importantly, the computational methods required for intelligent optimization. Signal processing methods are elaborated before exploring the benefits of traditional machine learning. Further improvements with deep learning are then presented, with fundamental theory explained. Relevant literature is briefly discussed, while a deeper literature review is provided in Chapters 3-5 related to each specific topic. These chapters also contain the results of experimentation in pursuit of generalization and reduction of human involvement in intelligent fault detection:

Chapter 3 Feature Representations and Machine State Separation

- A system to detect different operating modes and states of operating machinery is developed to allow a program to classify similar data into separate tasks.
- The short time Fourier transform (STFT) is used to compute the spectrogram, which is then run through a neighbour search for spectrogram peaks. This process builds a fingerprint of the signal and captures valuable time-frequency amplitude data.
- SVD-based PCA is used to reduce the dimensionality of the data and feed it as an input to a GMM for unsupervised clustering, which segments the dataset by states of similarity.

- The model is shown to recognize different operating states, as well as healthy and faulty bearings. The model also works on datasets with deteriorating bearings that change signal signatures over time.

Chapter 4 Generalization of Deep Learning for Vibration-Based Intelligent Fault Detection

- A CNN architecture is proposed which focuses on generalization of IFD for faulty bearings and helps prevent model overfitting.
- The architecture uses a combination of stacking filters to increase receptive fields, as well as bottlenecking to force the model to generalize spectrogram features at the sacrifice of ultra-high precision.
- Testing is done on datasets with large train/test splits, and results show that the generalized model scores higher accuracy than competing models, all while maintaining performance on regular train/test splits.

Chapter 5: Automation of Deep Learning Using Transfer Learning and Task Similarity Measurements

- This chapter combines the work in Chapters 3 and 4 to create a program that automates much of the labour in building ML models.
- The program is broken down into 3 main modules:
    - Module 1 aims to segment the incoming data by similarity. It draws upon the work in Chapter 3 to preprocess, fingerprint, reduce dimensionality, cluster the data into groups.
    - Module 2 aims to measure the data similarity with existing datasets, and guide the training or selection of a model based on generalization principles explored in Chapter 3

    - Module 3 uses ensemble learning to form the final prediction output of the committee of models in the program.
- Testing was performed on 3 datasets, and although performance did not exceed expectations, it still performed with +/-15% of benchmark averages, while greatly reducing the time to build, train, and deploy models.

In Chapter 6, conclusions are drawn, and future improvements and research directions are discussed in Chapter 7.

Chapter 2

Theory and Literature Review

This chapter will review the bulk of the theory needed to construct the solutions outlined in Section 1.3. Additional literature related to each element of the method will be presented, as appropriate. Therefore, Chapters 3, 4, and 5 will provide an additional review of the existing literature as necessary.

2.1 Maintenance Management Strategies

Strategic maintenance management focuses on the optimization of equipment usage, as well as personnel interactions with it [14]. In terms of equipment use, the goals are to enhance equipment reliability, condition, productivity/production, energy consumption, installation/operation, and environmental protection. In terms of personnel interactions, the goals are to optimize/enhance safety, personnel maintenance skills, and productivity. Sources may differ on the terminology and exact definition of types of maintenance strategies; however, the generally accepted categories are reactive maintenance, preventive maintenance, and predictive maintenance. Other specific categories, such as systematic maintenance, designed maintenance, and risk-based maintenance, exist but tend to be limited to certain industries or applications.

Reactive maintenance involves repairing or replacing damaged equipment once it has broken down. This form of maintenance is effective when the machine in question has a predictable failure pattern or causes minimal lost opportunity cost. In this case, the financial result of uptime is either non-mission critical or has limited risk in terms of operability of the project or company. Certain systems such as decision trees and maintenance management tools can improve the delivery of reactive maintenance, where the main goal is to save cost and reduce complexity. Reactive maintenance does, however, suffer in efficiency when the machine has a high uptime value, is a bottleneck, or is needed by other machines as part of a sub-routine. Specifically, if this machine experiences downtime, so does the production line, causing a large opportunity cost. In certain industries, such as automotive and manufacturing, opportunity cost of downtime in production can reach over $1M per hour [4]. Reactive maintenance also suffers if maintenance and repair networks must visit locations on site. Cost to fix, as well as first time fix rates, suffer greatly due to the time required to react and fix the issue.

Preventive maintenance, otherwise known as historic or periodic maintenance, is another maintenance category and aims to leverage regular inspection and tuning of equipment as a method to minimize the rate of failure [14]. These routines are constructed around the specific failure history of the equipment, as well as theoretical life cycles according to the machine and application in question. Reliability estimates, such as Weibull and exponential functions, can be employed to establish the maintenance schedule. Improvements are obtained by reducing the need to rely on reactive maintenance through avoiding faults altogether. However, this may result in over-maintenance, in which resources are inefficiently used to service equipment unnecessarily. In the case of clearly predictable and noticeable failure patterns, preventive maintenance can avoid faults. However, the risk of

catastrophic failure is still present in the event of periodic maintenance inaccuracies or errors.

Condition-Based Maintenance (CbM), also known as predictive maintenance, incorporates information about the equipment, along with prior knowledge of failure events, to monitor the health status of the equipment [15]. The goal is to diagnose, and predict which fault will occur, and ideally, estimate when it will occur. This is correlated with the scheduling of maintenance, allowing for the optimization of equipment lifecycles and utilization through the introduction of remaining useful life (RUL) or mean time to failure (MTTF) estimates. The primary activity of predictive maintenance is the collection of machine operational data and relating that data to historical analysis and the analysis of similar equipment to predict which component will fail and when, according to historical, present, and future equipment conditions. The advantage of predictive maintenance is the potential for optimizing the cost ratio per number of failures. The equipment is serviced before failures occurs, and health status monitoring avoids over-servicing and introduction of unnecessary costs. Some disadvantages of this strategy involve the implementation costs of required monitoring equipment, as well as the knowledge and technology barrier to executing this maintenance category correctly. Improvements to sensor technology and machine learning are gradually reducing these barriers and could make this management category increasingly accessible for a wider range of equipment.

As detection technology and reliability measures improve, the entire ownership model can be changed. For example, equipment-as-a-service (EaaS) models have been emerging recently: in this model, the equipment owner/operator pays based on product usage, rather than individual breakdowns or an upfront equipment cost. Fixed-price service contracts are also on the rise; where equipment owners pay a fixed price over a term of protection or maintenance [16]–[18]. New technology in both processing and sensing allows for improved

monitoring of the equipment and could ultimately lead to a complete understanding of the machine's health and operational state, known as a digital twin. However, challenges in data collection and connectivity to the machines must be addressed. For example, manufacturing equipment may often be installed in difficult to access locations, with limited to no access to a wireless connection, while cellular connections can introduce significant unwanted monthly costs. Security and privacy concerns are also a key issue, as commercial/industrial equipment can be of high value and may represent highly important applications. In addition to the challenges of connectivity, the computational requirements can be complex and introduce their own challenges. Sensor readings from historical data can be used to paint an understanding of a signal's failure. However, failures can be rare and are unwanted. Therefore, data scarcity and class imbalance are complexities that affect the proper execution of sensor-driven predictive maintenance programs.

2.2 Intelligent Fault Detection for Bearings

Knowledge of the presence of a fault within the subcomponents of a rotating machine is the key to enabling an efficient PdM program. The higher the degree of certainty and the higher the accuracy of that prediction results in a more confident and optimized program. False negatives could mean changing a bearing that didn't need to be changed, and false positives could mean accidentally running a bearing until failure, which could be costly and dangerous.

The earlier a fault can be identified in a machine, the earlier a fix can be addressed. Although there are other symptoms that a bearing is beginning to fail (e.g., increase in current draw, increase in component temperature, increase in audible noise), vibration is understood to be the earliest warning sign, and allows for the most amount of time to create

a reaction plan. This plan usually involves replacing the bearing, but depending on the size of the roller element, could mean identifying and tagging the fault and performing a repair. REBs can have different failure types, including inner race, outer race, ball, or cage faults (Figure 1).

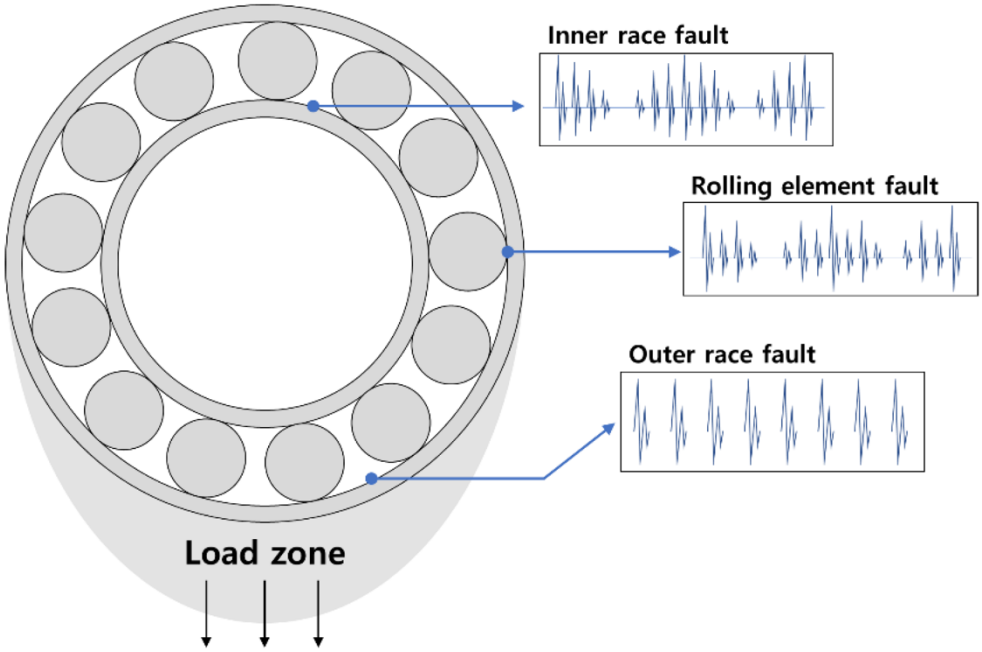


Figure 1: Location of REB faults and example of associated characteristic vibration signals [19]

With certain knowledge of the rotating equipment's specifications, fundamental fault frequencies, including the ball pass frequency outer (BPFO), ball pass frequency inner (BPFI), ball spin frequency (BSF), and fundamental train frequency (FTF), can be calculated using the following set of equations:

$$BPFO = RPM \cdot \frac{N_B}{2}\left(1 - \frac{B_D}{P_D}\cos(\beta)\right) \quad (1)$$

$$BPFI = RPM \cdot \frac{N_B}{2}\left(1 + \frac{B_D}{P_D}\cos(\beta)\right) \quad (2)$$

$$BSF = RPM \cdot \frac{P_D}{B_D}\left(1 - (\frac{B_D}{P_D}\cos(\beta)^2)\right) \quad 3$$

$$FTF = RPM \cdot \frac{1}{2}(1 - \frac{B_D}{P_D}\cos(\beta)) \quad 4$$

where $P_D = \frac{D_1 + D_2}{2}$, D_1 is the diameter of the inner race, D_2 is diameter of the outer race, B_D is the ball diameter, N_B is the number of balls, and β is the contact angle. By analyzing and tracking these frequencies, one can monitor the growth of specific fault bands, which lead to the identification of thresholds that could trigger a repair or replace decision. These equations are a function of the rotating speed of the equipment, and so, are fundamentally tied to frequencies and time-based analysis.

Beginning in the 1970s, research that used vibration to detect faults in REBs focused on envelope analysis [20]. By the 1980s, there was an increase in time and frequency domain analysis methodologies, as well as spectral analysis, which were applied to bearing and gear faults [21], [22] . Then, in the 1990s, there was a preliminary development and use of machine learning for improving the prediction accuracy and robustness of fault detection-based on vibration [23]. Neural networks and principal components analysis were among some of the first machine learning tools applied to the data that yielded promising results. Finally, the 2000s saw a dramatic growth in the number of publications and techniques used by machine learning algorithms, with the last decade focusing on deep learning techniques and new emerging strategies using generative adversarial networks and transfer learning methodologies [10].

2.3 Signal Processing

Performing IFD using mathematics and signal processing involves the identification of signal characteristics in either the time or frequency domain that may be associated with

faulty behaviour. Typically, this involves setting thresholds or ranges based on the analysis of a set of equipment's historical vibration data. One of the fundamental methods of categorizing a fault signal is through time domain analysis. Time domain simply refers to a signal representation of acceleration, velocity, or amplitude, plotted over time. Key time domain features involved in this analysis are shown in Table 1.

Table 1: Time domain features associated with IFD for REB [24]

Name	Formula	Eq #
Root Mean Squared	$\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$	5
Kurtosis	$\frac{1}{n}\sum_{i=1}^{n} \frac{(x_i - \bar{x})^4}{\sigma^4}$	6
Skewness	$\frac{1}{n}\sum_{i=1}^{n} \frac{(x_i - \bar{x})^3}{\sigma^3}$	7
Peak to Peak	$\lvert x_{min} - x_{max} \rvert$	8
Crest Factor	$\frac{\max \lvert x_i \rvert}{RMS}$	9
Shape Factor	$\frac{RMS}{\frac{1}{n}\sum_{i=1}^{n} \lvert x_i \rvert}$	10

Monitoring these features can help in identifying anomalies within the signal sample, and potentially help point out growing concerns. Elaborating these trends across time or a volume of signal samples can provide an overall view of the system's health state and aid in assessing if a fault is developing. This can be used to compare with historical data, or data from a similar environment, to help classify whether a fault is present or not. One of the challenges with time domain analysis is the indistinguishability between sources of vibration occurring at different frequencies. This makes it hard to separate noise, or other inputs that should or should not be considered, from the signal being analysed.

Frequency domain analysis can further investigate the relationship between the frequency components within the signal sample. Conversion between the time domain and frequency domain can be accomplished via a mathematical transform known as the Fourier transform, or the discrete Fourier transform (DFT) when discussed in the context of digital signal processing (the result of a data acquisition system sampling the voltage of an accelerometer at a given sampling rate). This sampling rate, f_s, serves as the base level at which information about the signal can be captured. A faster sampling frequency captures more intricacy about the system at the expense of the memory and computational cost required for storage and processing. The minimum required sampling frequency is set by the Nyquist theorem, which requires a sampling frequency of at least two times that of the highest frequency needing to be analyzed in the target signal:

$$f_s \geq 2f_h \tag{11}$$

where f_h is the highest frequency to analyze. Maintaining a sampling frequency above the Nyquist theorem is important to prevent aliasing, in which a higher frequency signal becomes misrepresented [25]. Conversion to the frequency domain is carried out using an optimized application of the DFT, known as the fast Fourier transform (FFT). The FFT on a discrete signal $x[n]$ of length N is given by:

$$y[k] = \sum_{n=0}^{N-1} e^{-2\pi j \frac{kn}{N}} \cdot x[n] \tag{12}$$

where $k = 0, \dots, N-1$. The frequency spectrum can then be plotted and further investigated by breaking down the frequency intervals and monitoring frequency domain features within those intervals. Figure 2 is an example of the power spectrum of an industrial motor. The frequency amplitude peaks are observable at constant intervals, which are harmonics of the fundamental frequency.

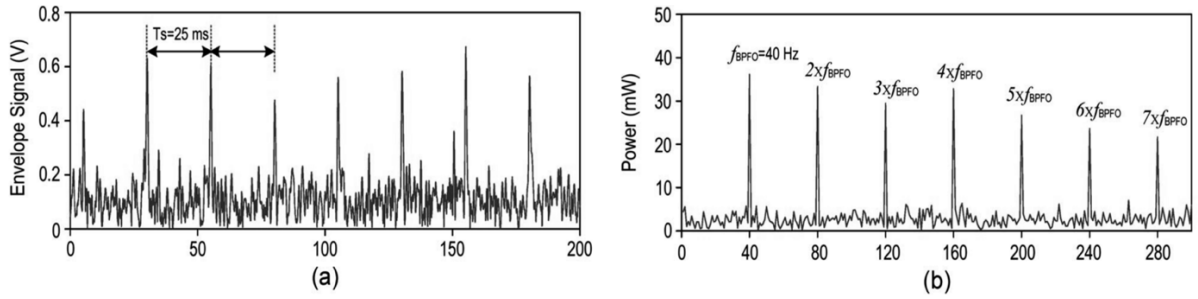


Figure 2: Observable faults in (a) the time domain in milliseconds, and (b) the frequency domain in hertz [26]

This is particularly relevant when dealing with rotating equipment, as the fundamental frequency can be set by the shaft rotation of the equipment. By analyzing the harmonics of this shaft rotation, specific symptoms of the equipment overall can be observed – this is a key strategy in the time-frequency analysis of motors and pumps. Evidently, if the fundamental frequency is not known, it poses a significant challenge to accurate IFD. Cepstrum analysis can be used to solve this problem by identifying the fundamental frequency and providing a benchmark for further analysis [27]. The Cepstrum of a signal can be calculated using:

$$C\big(x(t)\big) = F^{-1} \cdot \log(F[x(t)]) \qquad 13$$

where *F[x(t)]* represents the DFT. This operation takes a log amplitude spectrum and calculates its own spectrum (essentially creating a spectrum of a spectrum). This allows for pitch detection, where the first peak in the cepstrum represents the first harmonic, which can be used to identify the harmonic structure in the original signal, and thus, the potential fundamental frequency. Cepstrum analysis can also be used to calculate the mel-frequency cepstrum coefficients (MFCCs), which can be used as feature variables to monitor these harmonics. Figure 3 shows the process for extracting the MFCCs.

Figure 3: Process for extracting the MFCCs [28]

Having used equations for the DFT and for calculating the log amplitude spectrum, mel scaling is used to convert the time-frequency relationship to a logarithmic mel scale, which changes the perceptional relevance of frequencies. The conversion is defined as

$$f_{mel} = 2595 \cdot \log\left(1 + \frac{f}{500}\right) \qquad 14$$

where f is the frequency in Hz and f_{mel} is the frequency in mels. Once the frequency has been scaled, the discrete cosine transform is applied to get values that represent the formants and frequency content of the signal. The discrete cosine transforms return only real valued coefficients, rather than also returning complex values, such as in the case of the Fourier transform. The first coefficients are typically kept as they contain the most amount of information about the formants and spectral envelope. Traditionally, between 12 to 13 coefficients are kept when considering the MFCC values.

The time and frequency techniques mentioned serve to identify specific signal features that are typically associated with faulty vibration symptoms. Early success was shown, and some methods continue to be actively used today [4]. However, these techniques can be vulnerable to variations in the equipment and the outside world, including the equipment interacting with an external environment and having transient equipment operation cycles. As such, these techniques require custom development and deployment, which can be time consuming and costly. Signal processing techniques also fail to capture the small intricate details of the signal that are not apparent through traditional computational techniques, since rules must be explicitly programmed. This results in complications when changing sensor configurations or types of sensors altogether, as pointed out by Gao et al. [29].

Furthermore, the thresholds that qualify a signal as problematic are not often readily available for all equipment types given their operation. Finding the exact values that characterise a faulty bearing for each feature can be challenging to quantify.

2.4 Improvements with Machine Learning

Machine learning enables an algorithm to draw correlations in a set of data from examples, rather than by being explicitly programmed. In the context of IFD, this allows a computer to make correlations that go beyond the complexity that a human is able to compute, and removes the requirement for domain specific technical engineering knowledge when tuning the algorithm. At a high level, machine learning can be broken up into three main categories: unsupervised, supervised, and reinforcement learning. These categories contain algorithms that are specialized for accomplishing specific tasks, including regression, classification, dimension reduction, or clustering. This thesis will leverage unsupervised learning for the purpose of dimension reduction and clustering, as well as supervised learning for the purpose of classification. Reinforcement learning will not be addressed, as it is not within the scope of this thesis. Figure 4 shows a shortlisted set of popular machine learning algorithms in the context of predictive maintenance.

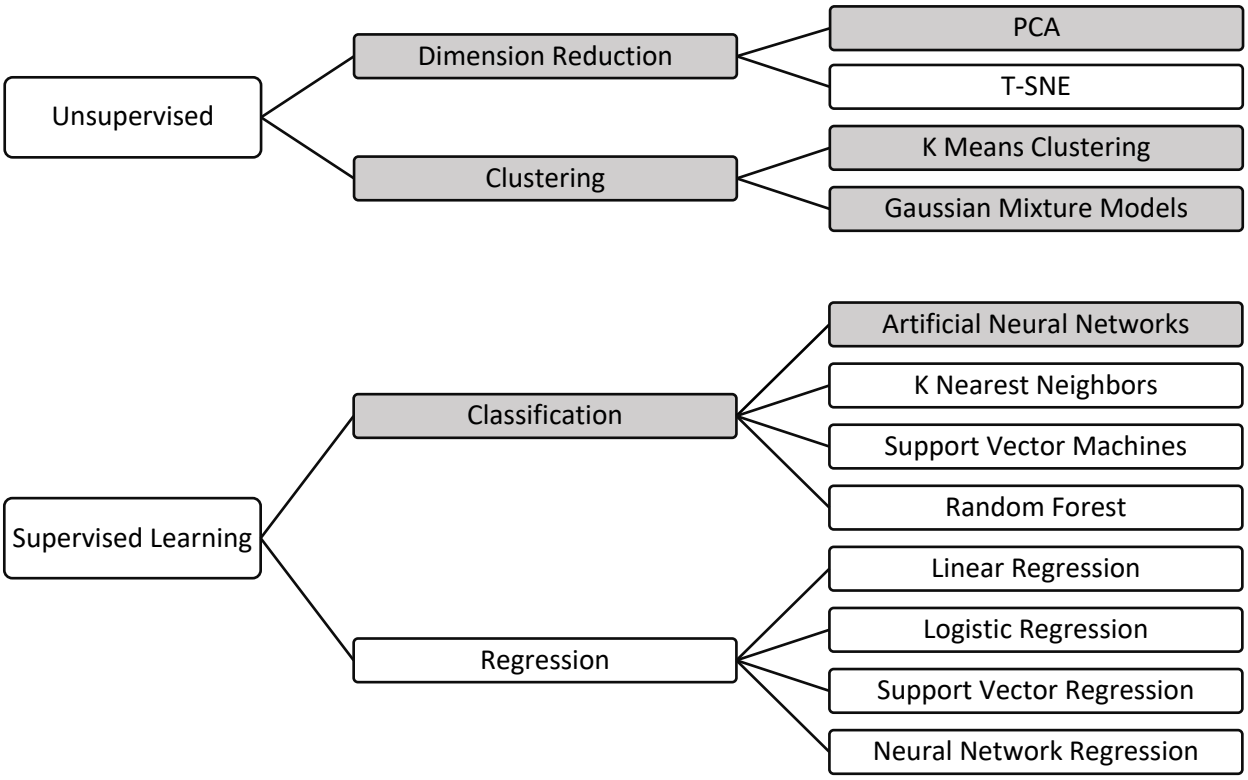


Figure 4: Popular algorithms in predictive maintenance literature (algorithms shaded in grey are used in this thesis) [10], [30]

As far as algorithm selection, there is heavy debate regarding which ML algorithm to use. In the context of classification, Brownlee et al. [31] suggest breaking down the classification problem by number of classes. If a multiclass classification problem exists, decision tree and random forest-based approaches tend to be recommended, such as the work conducted by Sun et al. [32] and in Saravanan et al. [33], who were able to achieve a 98% classification accuracy when combining wavelet features and a decision tree classification structure.

In most intelligent fault detection cases, the classification challenge has four classes or less. Usually, these include inner race fault, outer race fault, ball fault, or healthy. Some only use two classes: healthy or faulty. Only two classes are usually required in practice because if there is a fault in the bearing, the entire bearing will typically be replaced anyway. However, knowing the fault could help in prognostic estimations for remaining useful life,

since different elements of the bearing may degrade at different rates of time. Some also use severity of the fault as a class: no severity (healthy), mild, and severe. These classes can be associated with an actual metric or measurement, as was done with the Case Western Reserve University dataset that provides data for different fault sizes in mm of fracture [34].

2.4.1 Supervised Learning and Artificial Neural Networks

In supervised learning, the data is generally in the form of (x_i, y_i), where x_i is the feature vector and y_i is the label distinguishing its true state. Two of the most common categories of supervised learning involve classification or regression [35]. Regression problems tend to focus on continuous variables involved in the prediction of consumption, stock prices, and any other fluctuating variable, often over time. Given that this thesis focuses on the identification of faulty bearings, classification algorithms are considered herein, since classification of a faulty bearing is a method of performing IFD. Regression could be added to estimate the remaining useful life of the bearing given the fault. However, that is not within the scope of this thesis. In classification, the model's output is the y_i label for the purpose of classifying the data, whether that be a binary classification or a multi-class classification with more than two classes. A false negative is defined by the case where the algorithm incorrectly predicts a healthy bearing, and a false positive is when the algorithm incorrectly predicts a faulty bearing, with true positive and true negative being the respective correct predictions. This terminology can then be used to establish a set of metrics:

$$Accurracy = \frac{1}{n}\sum_{i}^{n}(\hat{y}_i = y_i) \quad 15$$

$$Precision = \frac{N_{tp}}{N_{tp} + N_{fp}} \quad 16$$

$$Recall = \frac{N_{tp}}{N_{tp} + N_{fn}} \quad 17$$

$$F1\ Score = 2 \cdot (\frac{Precision\ \times\ Recall}{Precision + Recall}) \quad 18$$

where n is the number of samples, $\hat{y}_i$ is the predicted class, y_i is the actual class, N_{tp} is the number of true positives, N_{fp} is the number of false positives, N_{fn} is the number of false negatives. Given these metrics, artificial neural networks (ANNs) are used for making supervised classification in this thesis. ANNs form the basis for deep learning and are a sub-field of machine learning. ANN algorithms are loosely inspired by the structure of the human brain. The overall structure can be described as an input layer that receives the incoming data, hidden layers that perform computations based on non-linear activation functions, then an output layer that predicts the final output, as shown in Figure 5.

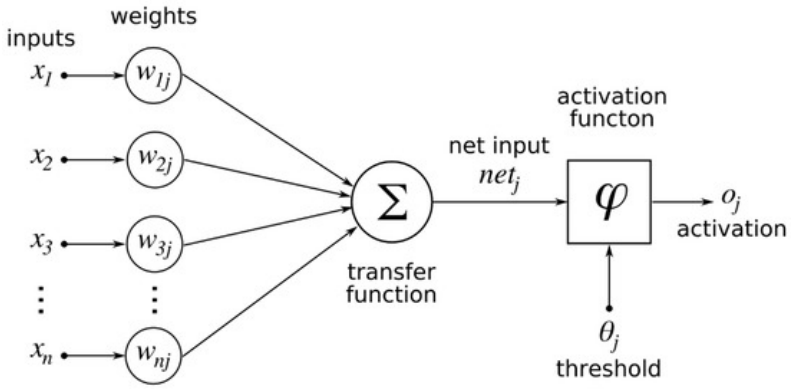


Figure 5: Visual representation of an artificial neural network [36]

Each neuron in the input layer is connected to neurons in the following hidden layer via weights $w_{n,i,j}$, where n is the layer number, and i,j are the indexes of the connecting neuron in the current and next layer, respectively (note: in Figure 5 there is only one neuron so there is no layer number). Each neuron is associated to a numerical value called the bias, which is

then added to the input sum. The weights are multiplied with the inputs, then the bias is added, and their sum is sent as an input to the neurons in the hidden layer. Input data for a given neuron in the hidden layer would be the sum of the previous layer's neuron values multiplied by its associated weight. For example, the value of the first neuron in the first hidden layer, given by $a_{1,1}$, would be:

$$z_{1,1} = w_{1,1,1}x_{1,1} + w_{1,2,1}x_{1,2} + \cdots + w_{1,i,1}x_{1,i} + b_{1,1} = \sum_{i=1}^{n_0} w_{1,i,1}x_{1,i} + b_{1,1} \qquad 19$$

where $w_{1,1,1}$ is the unique weight associated to the input layer's first neuron $x_{1,1}$. All this information is then summed together, with the addition of a bias $b_{1,1}$, for every neuron. Note that most neural networks are composed of multiple layers and neurons, like in Figure 6.

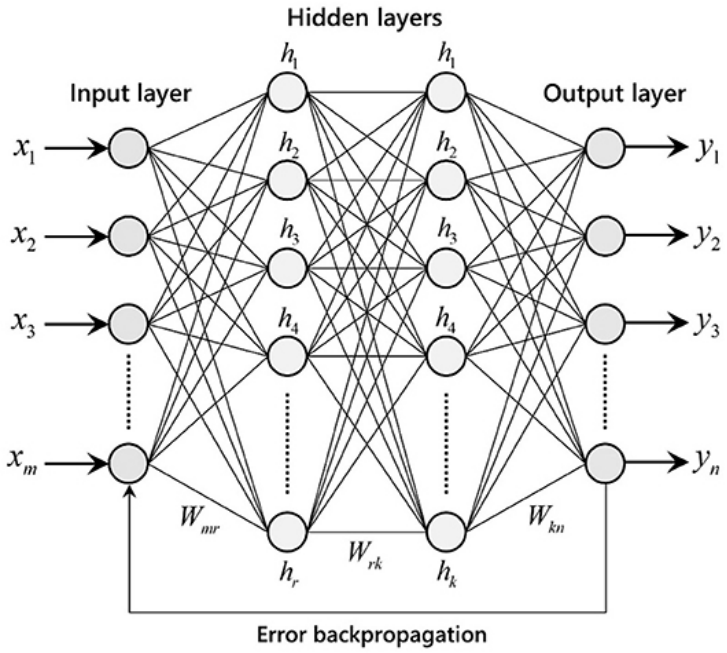


Figure 6: An example of an artificial neuron network with multiple layers and neurons [37]

Generalizing this for any number of layers in a model gives:

$$z_{i,j} = \forall_{j=1}^{n} \sum_{i=1}^{n_0} w_{n,i,j}x_{i,j} + b_{n,j} \qquad 20$$

The value is then passed through a threshold function called the activation function. The activation function determines whether a neuron will get activated or not (i.e. whether a neuron transmits data to the neuron of the next layer in the channel). There are two common activation functions used, either the rectified linear unit (ReLu) or the sigmoid function. Both are described in Figure 7.

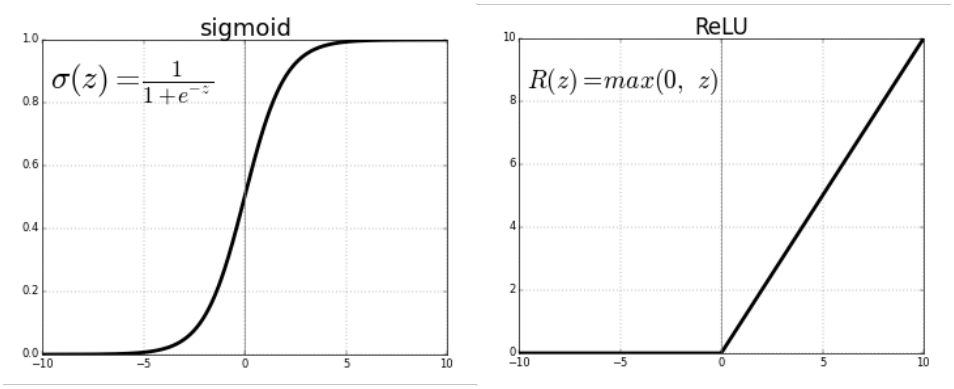


Figure 7: Visual representation and equations for sigmoid and ReLu activation functions [38]

Passing the value through the activation function gives:

$$R(z)_n = \forall_{j=1}^{n} \max\left(0, \sum_{i=1}^{n_0} w_{n,i,j} x_{1,j} + b_{n,j}\right) \quad 21$$

The last layer, the output layer, uses a softmax activation function that converts the vectors into a probability distribution between 0 and 1. The sum of all real numbers in the softmax layer is 1, which allows them to be interpreted as probabilities.

$$\hat{y} = softmax(z_n) = \forall_{j=1}^{n} \frac{e^{z_{n,1}}}{\sum_{k=1}^{n} e^{z_{n,k}}} \quad 22$$

This flow of information from input layer to output layer, given the rules, is the forward propagation process. This process allows the model to ingest an input, and compute a corresponding output dictated by the values of the neurons, weights, and biases. The goal of backpropagation is then to measure the error between the predicted output and the ground

truth, and relate that error to changes in the weights and biases in the network. In more formal terms, backpropagation seeks to compute the partial derivative of the cost function with respect to the tunable hyperparameters w and b given as $\frac{\partial C}{\partial w_{j,k}^{l}}$ and $\frac{\partial C}{\partial b_{j,k}^{l}}$, respectively. The cost function is computed using:

$$L = \frac{1}{2}\sum \|y(x) - z^{L}(x)\|^{2} \qquad 23$$

There are two assumptions that need to be made: the first is that individual samples of x can be averaged over the cost function, and the second is that the cost function can be written as a function of outputs from the network.

The first step in backpropagation is to relate the error to the change in weights and biases. Suppose Δz_j is measured as the change to a neuron's input on a given layer. This results in an expression defining the error of neuron j in layer l given as

$$\delta_j^l = \frac{\partial L}{\partial z_j^l} \qquad 24$$

Now, the vector δ_j^l must be computed for every layer and those errors must be related to $\frac{\partial C}{\partial w_{j,k}^{l}}$ and $\frac{\partial C}{\partial b_{j,k}^{l}}$, given by the function:

$$\delta_j^L = \frac{\partial L}{\partial a_j^L}\sigma'(z_j^L) \qquad 25$$

where $\frac{\partial C}{\partial a_j^L}$ is the rate of change of the cost function of the *jth* activation and $\sigma'(z_j^L)$ measures the rate of change of the activation function at z_j^L. Note that for small rates of change, the vector δ_j^L will be small. This means that higher relative proportions in change lead to the most rapid decrease in the cost function; there is a saying: *"neurons that fire together wire together"*. The process of forward propagation to calculate the output and backpropagation to tune the hyperparameters can be a lengthy process, so a stochastic gradient descent can be

used to accelerate this process by randomly shuffling training data and computing the gradient descent on smaller batches of data. This results in slightly less accurate, but much larger steps in cost minimization.

Early applications of ANNs for predictive maintenance and intelligent fault detection include work conducted by Alguindigue et al. in which fault diagnosis was performed on RED vibration data collected from Electricite de France [23]. The data had 8 pattern categories, each representing an operating state of the equipment. The ANN classifier correctly predicted 90% of the categories. However, the authors noted the network's challenges with false positives; most incorrectly classed patterns were from damaged bearings that the network perceived as good. The authors also noted performance challenges and inconsistent operation conditions. Later, Paya et al. improved on the base ANN by combining it with wavelet transforms as an input preprocessor. They tested their algorithm on roller element bearings in a gearbox, electric motor, bearing housings, and disc brake. In a laboratory setting, the authors were able to categorize the vibration signals into 6 distinct groups with an accuracy of 97%. The authors still did note challenges in computation time and speed, as well as configuration variability [39]. More recent works involve combining ANNs with other classifiers, such as SVM, like in the work of Kankar et al. [40], where the authors proved that this combination resulted in performance improvements over using SVM as a classifier alone.

Despite ANNs and multilayer perceptrons (MLPs) showing improvements over traditional signal processing methods, one of the major challenges is the customization of hyperparameter configurations that can be costly in both time and effort. Moreover, accurately labelled data is a fundamental requirement for building solid models. Unfortunately, accurately labelled data is scarce, especially for breakdown events that many are actively trying to avoid. This is where unsupervised learning can be beneficial.

2.4.2 Unsupervised Learning for Dimension Reduction and Clustering

Unsupervised learning, in the context of this thesis, uses algorithms that reduce data volume or discover patterns in the data provided, without being given a ground truth via labelled data. In this case, unsupervised learning is specifically used for clustering and dimension reduction. For dimension reduction, features are reduced in space while trying to capture as much variance as possible. Here, variance is defined as the expected deviation from a given sample in comparison to its mean [41]:

$$S_j^2 = \sum_{i=1}^{N} \frac{(x_{ij} - \bar{x}_j)^2}{N} \tag{26}$$

where x_i is the data point, $\bar{x}_j$ is the sample mean, and N is the number of samples. The covariance is then a measure of the joint variation between the *j-th* and *k-th* variables:

$$S_{jk} = \sum_{i=1}^{N} \frac{(x_{ij} - \bar{x}_j)(x_{jk} - \bar{x}_k)}{N} \tag{27}$$

The covariance summarizes the relationship between samples by identifying either positive trends, negative trends, or no apparent trends. Essentially, it maps how one variable changes with respect to another. These equations and insights can be used to capture the variance between all pairs of components represented by R_{jk}, and form a correlation matrix, given as:

$$Corr = \begin{bmatrix} 1 & \cdots & R_{1k} \\ \vdots & \ddots & \vdots \\ R_{j1} & \cdots & 1 \end{bmatrix} \tag{28}$$

where,

$$R_{jk} = \frac{S_{jk}}{S_j \; S_k} = \frac{\sum_{i=1}^{N}(x_{ij} - \bar{x}_j)(x_{ik} - \bar{x}_k)}{\sqrt{\sum_{i=1}^{N}(x_{ij} - \bar{x}_j)^2}\sqrt{\sum_{i=1}^{N}(x_{ik} - \bar{x}_k)^2}} \quad 29$$

There are several measurements that can be used to distinguish distances when referring to variance or other forms of measurement in dimension reduction and clustering, as provided in Table 2. These distance measures are important because they are the metrics used for grouping data into clusters. Dimension reduction first serves to reduce the total data size by removing unnecessary information while still preserving variance.

Table 2: Types of distance measures and their equations

Name	Formula	Eq #
Euclidean	$d_{euc} = \sqrt{[(x_2 - x_1)^2 - (y_2 - y_1)^2]}$	30
Manhattan	$d_{man} = \lvert x_1 - x_2 \rvert + \lvert y_1 - y_2 \rvert$	31
Minkowski Distance	$d_{mink} = (\sum_{i=1}^{n} \lvert x_i - y_i \rvert^p)^{1/p}$	32

In the case of vibration based IFD, features of each sample are calculated and appended to form a feature vector of length m, given by $\hat{x}_n$, where each element is a corresponding statistical result. These feature vectors are then appended together to form a matrix containing the features of each sample. This matrix is defined as the dataset matrix (or the sub-dataset matrix if there are multiple configurations that should not be grouped in the same matrix). This matrix X_D of shape $(m \times n)$ is defined as [42]:

$$X_D = \begin{bmatrix} | & | & & | \\ \hat{x}_1 & \hat{x}_2 & \dots & \hat{x}_n \\ | & | & & | \end{bmatrix} \qquad x_k \in \mathbb{R}^n \quad 33$$

With n being the number of samples (which is equal to the number of feature-vectors). The dataset matrix X_D can then be decomposed as the product of three matrices U, Σ, V as follows:

$$CX_D = U\Sigma V^T \tag{34}$$

Where U and V are orthogonal matrices of size $(m\ x\ m)$ and $(n\ x\ n)$, respectively, and Σ is a diagonal matrix of decreasing singular values σ_n. The resulting matrix X_D is given as:

$$X_D = U\Sigma V^T = [U_1, U_2, \dots, U_n]\begin{bmatrix}\sigma_1 & \cdots & 0\\ \vdots & \ddots & \vdots\\ 0 & \cdots & \sigma_n\end{bmatrix}[V_1, V_2, \dots, V_n] \tag{35}$$

$$X_D = \sum_{i=1}^{n} U_i \sigma_i V_i \tag{36}$$

where U and V are unit matrices, such that:

$$UU^T = U^T U = \mathbb{I}$$

and $\mathbb{I}$ is the identity matrix. Matrix Σ is a diagonal non-negative matrix ordered by magnitude. Therefore, the first column of U and V, both corresponding to σ_1, are meaningfully more important than the second columns, and so on. The relative importance of the columns is given by the singular values in sigma. This leads to the ability to negate the less important values of sigma, which are associated to less important columns of U and V, leading to a significant reduction in the matrix size while preserving the variance and information captured. This provides a basis for PCA, which is used to reduce the dimensions of data being used. T-distributed stochastic neighbour embedding (T-SNE) can also be used to reduce the dimensionality of data. The main difference between PCA and t-SNE is that t-SNE's mappings are local and based on neighbourhoods of points, statistically determined to be relevant to each other, whereas PCA is a projection of eigenvectors based on the diagonal rotation of the covariance matrix (i.e. global).

In most studies, dimension reduction is usually combined with some form of clustering in order to achieve a classification [10]. Clustering allows grouping data based on similarity or differences in the variance, or some other distance measurement discussed above. Clustering can be exclusive, probabilistic, overlapping, or hierarchical [35]. K-means is an example of

exclusive clustering, where data points are assigned to clusters based on the minimization of variance, as given by

$$V = \sum_{j=1}^{J} \sum_{k=1}^{K} r_{jk} \left\| x_j - \mu_k \right\|^2 \qquad 37$$

where cluster centers $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ are randomly initialized. Through iteration of every i set, groups are formed that minimize the variance invoked by each element within a cluster such that the sum of the squares of the distance, $\left\| x_j - \mu_k \right\|^2$ in order to minimize V, given by

$$r_{jk} = \begin{cases} 1 \quad if\ k = argmin_j \ \left\| x_j - \mu_k \right\|^2 \\ 0 \quad otherwise \end{cases} \qquad r_{jk} \in \{0,1\} \qquad 38$$

This results in mapping points to their nearest associated centroid, all while selecting centroid locations that result in the least possible variance across all points. Contrary to this, fuzzy logic interpretations involve overlapping clusters of data points with varying relationships to a given cluster. Meaning, the relationship of a point to a given cluster is defined by a probability, rather than an association of 0 or 1. Gaussian mixture models (GMMs) are an example of probabilistic clustering in which a distribution is leveraged to assign a probability that a given data point belongs to a particular cluster. An expectation maximization algorithm is often used to estimate the probability of this given relationship.

In predictive maintenance and IFD, a valid reason for using GMMs is their ability to account for overlapping data with potentially missing or unexpected points. These soft classification methods have been shown to be superior to K-means methods for their ability to deal with potentially unevenly grouped subpopulations of data [43]. However, one of the challenges faced by GMMs is the selection of the number of Gaussians to use (how many centroids). Interesting work has been done to address GMMs having no prior knowledge, as well as the optimization of the number of clusters the algorithm forms in the expectation-

maximization (EM) process [44]. A more detailed literature review on this topic will be conducted in Chapter 3.

2.5 Further Improvements with Deep Learning and Transfer Learning

So far in this thesis, signal processing has been discussed for identifying the characteristics of vibration signals that represent faulty bearings. However, traditional signal processing methods struggle to adapt to complex signals that interact with the environment and its' noise. Furthermore, signal processing solutions need to be tuned for specific equipment arrangements based on ground rigidness, operation modes, and other factors. Machine learning algorithms improve on this by replacing some manual tuning requirements with the training of hyperparameters through labelled data, allowing a machine learning model to make better predictions without being explicitly programmed. However, challenges with data volume and variety are still present, as well as performance issues related to scalability. As datasets grow in volume, the variability between data classes increases, and so does class imbalance (i.e. an abundance of healthy data, but low amounts of faulty data). This class imbalance can have a large impact on the prediction output of a machine learning algorithm. Also, basic machine learning algorithms cannot take advantage of 2D inputs such as spectrograms. Spectrograms reveal relationships within the signal in the time-frequency domain, which is valuable information for a neural network. Deep learning architectures such as convolutional neural networks (CNNs) are uniquely positioned to take advantage of higher dimension inputs as well as capture more intricate details for non-linear relationships within large datasets. The use of deep learning and transfer

learning, in the context of this thesis, provides performance advantages over traditional machine learning algorithms.

2.5.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of deep learning architecture that specialize in 2D or 3D inputs, usually in the form of images. This is because traditional ANNs don't consider spatial information or neighbouring pixels, whereas CNNs make use of spatial information by using convolutional and pooling layers. Usually, image data has some form of structure (edges, shapes, textures), and images usually scale (i.e. a square is a square regardless of how big it appears in an image). So, CNNs capture image content by extracting basic features (bars, curves, etc.) via kernels. Kernels are a grid of weights applied to the pixel representation of the image. The kernels are overlayed on top of the image, and the dot product of the image segment values and the kernel values is calculated. This yields a number on the output grid. After the dot product is completed, the kernel slides to the right and does the same thing until it reaches the end of the image, where it returns to the horizontal start position before sliding down and performing the same set of calculations left to right. The horizontal and vertical sliding steps are known as the stride, as shown in Figure 8.

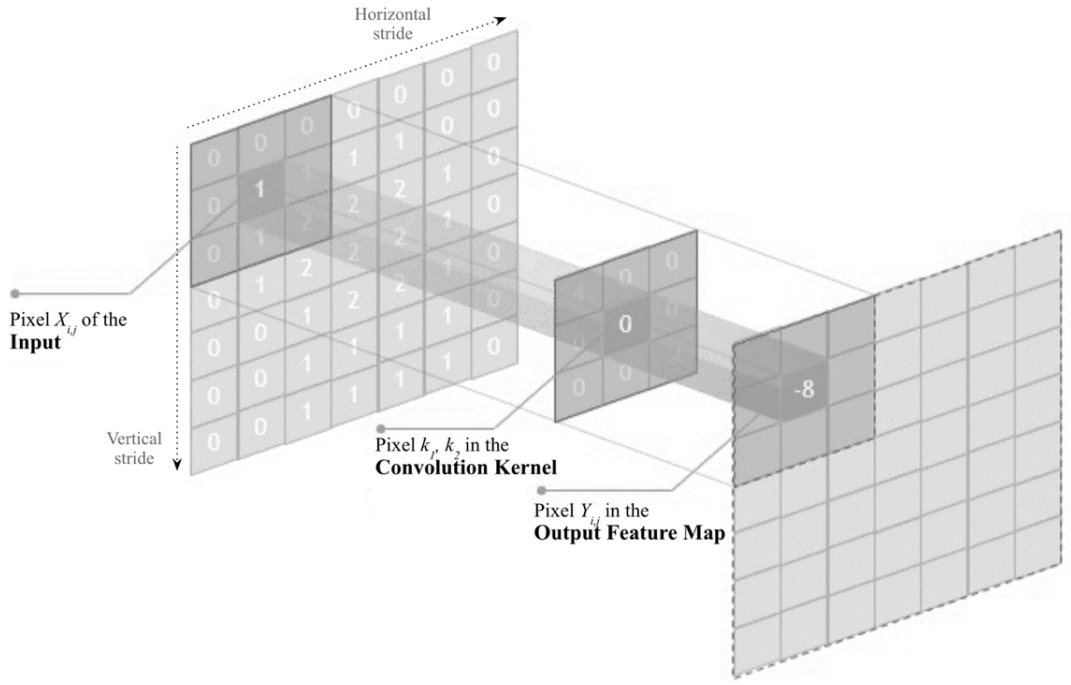


Figure 8: High-level CNN architecture [45]

Image edges do pose a challenge because the kernel cannot properly overlap with the side of the image. Therefore, an artificial edge of zeroes is added to the image, known as zero padding. This lets the kernel center itself on the edge of the image and capture all the information the image contains. For the purposes of this thesis, only greyscale images are used, so there is only one channel (C=1). Therefore, we assume the depth of our output map to be 1. The expression for the output feature map of the *lth* layer is given as

$$Y_{i,j}^{l} = \sum_{a=0}^{K_1-1} \sum_{b=0}^{K_2-1} X_{i+a,j+b}^{l-1} \cdot W_{a,b}^{l} + b^{l} \qquad 39$$

where X is the output at layer l, and i and j are iterators over the kernel dimension K_1 and K_2. $W_{a,b}^{l}$ is the weight matrix connecting neurons of *l-1* to l, and b^{l} is the bias at layer l.

Like the hyperparameter tuning process used with ANNs, CNN backpropagation looks to evaluate the partial derivative of changes in weights and biases with respect to the loss

function. Equation 40 represents the resulting change in the loss function based on changes in each weight kernel pixel $w_{a',b'}$ at layer l.

$$\frac{\partial L}{\partial W^l_{a',b'}} \tag{40}$$

Whereas equation 41 computes the resulting change in the loss function based on changes in each input pixel $X^l_{i',j'}$ at layer l.

$$\frac{\partial L}{\partial X^l_{i',j'}} \tag{41}$$

The iterative calculation of these gradients allows for the optimization of the loss function over a set of training samples going through forward and backwards propagation. Due to the nature of convolution, and the fact that kernel pixels affect all pixels in the output map, each pixel $w_{a',b'}$ will have an impact on every element of the resulting outputs. Using the chain rule on the individual weight gradient components results in a dual summation, expressed as:

$$\frac{\partial L}{\partial W^l_{a',b'}} = \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta^l_{i,j} \cdot X^{l-1}_{i+a',j+b'} \tag{42}$$

where $\delta^l_{i,j} = \frac{\partial L}{\partial Z^l_{i-a,j-b}}$ represents the output gradients in layer l, and H and W are the height and width of the image, respectively. For computing the gradient of the loss function with respect to an input pixel $X^l_{i',j'}$ at layer l, the chain rule can also be used, such that:

$$\frac{\partial L}{\partial X_{i',j'}} = \sum_{a=0}^{K_1-1} \sum_{b=0}^{K_2-1} \frac{\partial L}{\partial Y^l_{i'+a,j'+b}} \cdot W^l_{a,b} \tag{43}$$

Once again, this demonstrates that computations are saved as the partial derivations propagate backwards through the network.

Kernel dimensions K_1 and K_2 are usually square and sized as odd numbered grids to obtain a central value to use as a reference. However, even numbered and rectangular kernels can be used [46]. There can also be multiple kernels used on a layer, which will control the depth of the layer, as well as the corresponding depth of the output. Usually, CNNs have multiple kernels that can represent the different features detected. The values in the grids of these kernels are tuned just like the hyperparameters in an ANN. However, they act as feature detectors able to detect lines and edges. A further literature review on CNNs will be conducted in Chapter 4.

2.5.2 Transfer Learning, Gabor Filters, and Receptive Fields

Transfer learning is the practice of using the information learned from one dataset and applying it to a similar but different domain to achieve better results. For example, it could involve training a CNN on images of cats for the purpose of breed classification, and then using that CNN to perform a similar classification but on images of leopards. There are many frameworks that have been proposed for organizing transfer learning, but the Pan and Yang framework is one of the most popular [47]. Pan and Yang define transfer learning through the concepts of domains $\mathcal{D}$ and tasks $\mathcal{T}$, given by

$$\mathcal{D} = \{\mathcal{X}, P(X)\} \tag{44}$$

$$\mathcal{T} = \{\mathcal{Y}, P(Y|X)\} \tag{45}$$

where $P(X)$ is the marginal probability distribution, $X = \{x_1, \dots, x_n\} \in \mathcal{X}$, and $\mathcal{Y}$ is the label space with a predictive function $P(Y|X)$ such that $Y = \{y_1, \dots, y_n\} \in \mathcal{Y}$. If one source domain and one target domain are assumed, a data instance $x_{S_i} \in \mathcal{X}_S$ and the corresponding label

class $y_{S_i} \in \mathcal{Y}_S$, the source and target domain data can be defined as source domain $\mathcal{D}_S$ and target domain $\mathcal{D}_T$ given by

$$\mathcal{D}_S = \left\{\left(x_{S_1}, y_{S_1}\right), \ldots, \left(x_{S_{n_S}}, y_{S_{n_S}}\right)\right\} \quad 46$$

$$\mathcal{D}_T = \left\{\left(x_T, y_{T_1}\right), \ldots, \left(x_{T_{n_T}}, y_{T_{n_T}}\right)\right\} \quad 47$$

Pan and Yang provide a formal definition of transfer learning:

> *"Given a source domain $\mathcal{D}_S$ and learning task $\mathcal{T}$, a target domain $\mathcal{D}_T$ and learning task $\mathcal{T}_T$, transfer learning aims to help improve the learning of the target predictive function in $\mathcal{D}_T$ using the knowledge in $\mathcal{D}_S$ and $\mathcal{T}_S$, where $\mathcal{D}_S \neq \mathcal{D}_T$, or $\mathcal{T}_S \neq \mathcal{T}_T$."*

With this framework in mind, specific transfer learning methodologies can be established, as shown in Table 3.

Table 3: Transfer learning methodology proposed by Pan and Yang [47]

Transfer Learning Settings	Source Domain Labels	Target Domain Labels	Source and Target Domains	Source and Target Tasks
Inductive Transfer Learning	Available	Available	Same	Different but related
Transductive Transfer Learning	Available	Unavailable	Different but related	Same
Unsupervised Transfer Learning	Unavailable	Unavailable	Different but related	Different but related

In this thesis, the source domain and target domain labels are both available, but the target domain labels may be omitted for the purpose of testing real-world applications where target domain labels would not normally be available. This approach is known as inductive transfer learning (a branch of multi-task learning). A formal definition of multi-task learning is best given in a widely cited paper by Caruana [48]:

> *"Multitask Learning is an approach to inductive transfer that improves generalization by using the domain information contained in the training signals of related tasks as*

> *an inductive bias. It does this by learning tasks in parallel while using a shared representation; what is learned for each task can help other tasks be learned better"*

The concept of inductive transfer learning through multi-task learning hinges on a shared data representation between source and target domains. In predictive maintenance, this is semantically represented through similarities in the fault characteristics of the spectrogram, such as higher than normal energy levels at certain harmonics and sharp spikes at certain frequencies. The Kullback-Lieber divergence, which measures the difference in statistical distribution from one dataset to another, is a technique used to measure this similarity. The expression of probability distribution is given by:

$$D_{KL}(p(x)||q(x)) = \sum_{x \in X} p(x) \cdot ln \frac{p(x)}{q(x)} \quad 48$$

where x denotes the classes, and $p(x)$ and $q(x)$ are distributions over those classes. Note that when it comes to similarity measurements between datasets, PCA cannot be used since PCA is based on variance as opposed to distributions over the dataset. Should there be very low dataset similarity, there can be a negative impact on model performance when transfer learning [49].

In a widely cited study, Yosinski et Al. assessed what to transfer and when to transfer, providing the insight that transfer learning can be negatively impacted by the specialization of higher layer neurons, as well as possible difficulties in splitting networks between co-adapting neurons [50]. In the context of this thesis, this means that the goal of transfer learning in predictive maintenance is to ensure feature generalization of the shallow network layers, and fine tuning the network by training on the labels in the target domain. To properly promote this behaviour, the concepts of Gabor filters and receptive fields can be used. Many studies have compared the low-level filters in a CNN to Gabor filters, convolutional filters that combine gaussian and sinusoidal terms to analyze frequencies and detect frequency

content in specific directions (with the sinusoidal component providing directionality and the gaussian component providing the weighted value) [51]. This relationship is given by

$$g(x, y;\ \sigma, \theta, \lambda, \gamma, \varphi) = \ exp(\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}) \cdot \exp\ (i(2\pi \frac{x}{\lambda} + \varphi)) \qquad 49$$

where $x' = xcos\theta + ysin\theta$, and $y' = -xsin\theta + ycos\theta$. The Gabor is a function of the position x and y, as well as the standard deviation σ, the orientation of the filter θ, wavelength λ, and aspect ratio γ. By changing these parameters, different kinds of filters can be created that might detect certain features such as edges, textures, and simple shapes. Figure 9 shows a set of Gabor filters and their convoluted output on an input image.

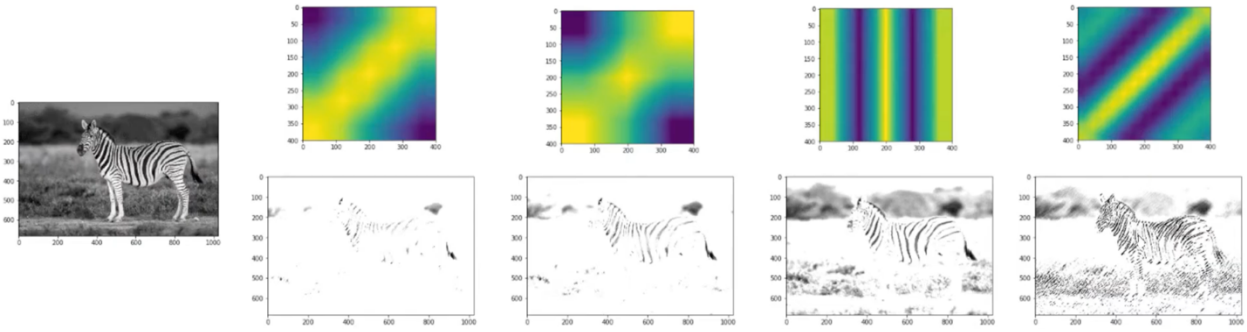


Figure 9: A visualization of various Gabor filter parameters and their outputs [52]

As the kernels in a CNN tune, they tend to resemble a bank of different Gabor filters. These filters are not unique to a specific domain, meaning, Gabor filters have properties that allow them to transfer information from the source domain to a target domain. Additional fully connected layers can then be used to transition the model's output from generic to specific in the final layers.

Improving the kernels' visibility in the network, and therefore improving the Gabor filter banks, leads to a better generalization of the lower-level layers in the deep network. In formal terms, the receptive field is an important consideration when training CNNs [53]. Receptive

fields can be used to determine the pixels that each convolutional layer can see at various layers in the model. This is defined by:

$$R_k = 1 + \sum_{j=1}^{k} (F_j - 1) \cdot \prod_{i=0}^{j=1} S_i \qquad 50$$

where F_j is the filter size of layer j, and S_i is the stride value of layer i. At layer k, the area defined by $R_k \times R_k$ is the input that each pixel of the kth activation map can see.

Consider a stack of three consecutive convolutional layers, as shown in Figure 10. In this case, a neuron in the third layer sees more that just a 3x3 view of the previous layer, but also a 5x5 view of the layer before that, and a 7x7 view of the initial input layer.

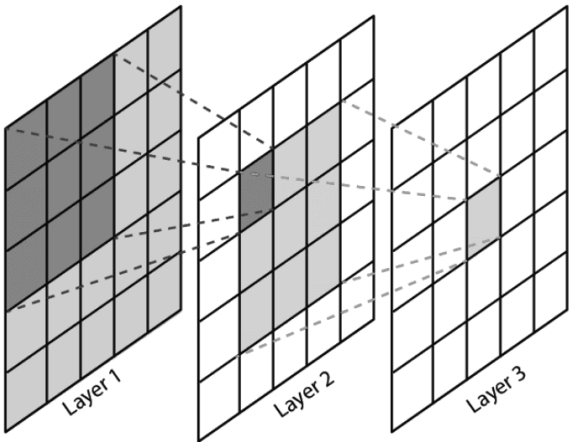


Figure 10: Visualization of pixel visibility throughout layers in a CNN [54]

The benefit of this setup over a single 7x7 filter is the ability to capture nonlinearities between the layers, compared to only having a single linear function. Doing this also subjects the network to fewer hyperparameters, as show in Table 4.

Table 4: Number of hyperparameters for single vs stacked filter configurations

Single Large Filter	**Stacked Smaller Filters**
$K_1 = K_2 = 7$, $C = 1$	$K_1 = K_2 = 3$, $C = 1$
$N_{Hyperp} = C \cdot (K_1 \times K_2 \times C) = 49$	$N_{Hyperp} = K_1 \cdot (C \cdot (K_1 \times K_2 \times C)) = 27$

In summary, stacking smaller filters allows more non-linear expression of a signal, while requiring fewer hyperparameters, at the expense of more layers and the additional memory required for backpropagation.

As far as transfer learning is applied to predictive maintenance, there are four main scenarios being considered in this thesis, described in Table 5. The distribution in the number of publications associated with each is shown in Figure 11.

Table 5: Application scenarios of transfer learning towards IFD for REB [55]

Transfer to an Identical Machine (TIM)	The same equipment is used for data collection in source and target domains, but usually under varying operating conditions.
Transfer across Different related Machines (TDM)	The source domain piece of equipment is different than the target domain, but is still mechanically similar. An example would be electric motors of different weights, sizes, speeds, etc. The failure modes are still similar or the same.
Transfer from Laboratory to Real Machine (TLRM)	The source domain would be a laboratory test rig and the target domain would be a similar but real piece of equipment. The difference with TDM is the emphasis on the target domain being a real-world piece of equipment.
Transfer from Virtual to Real Machine (TVRM)	The source domain is a virtual model or simulation, whereas the target domain is a real-world piece of equipment.

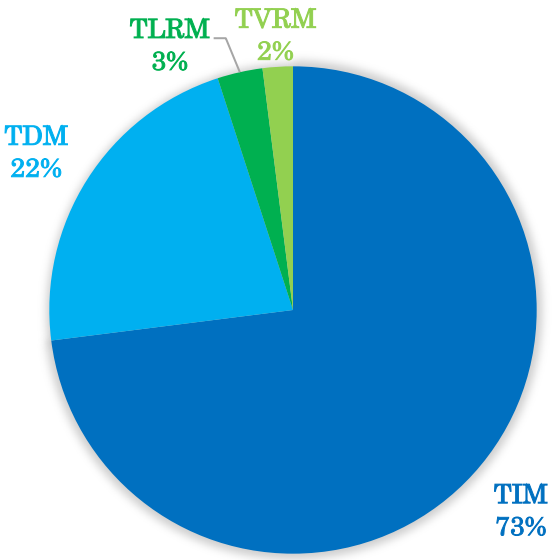


Figure 11: Distribution of existing literature on transfer learning case scenarios.

The reason TLRM and TVRM publication are so scarce is that they involve real pieces of equipment, which are exceptionally hard to come by in research. Therefore, the focus of this thesis will be on TDM, which is still significantly under researched in proportion to TIM. Furthermore, this thesis establishes a strategy that can be applied to TLRM, despite not yet being tested on it. Section 5.1 will review more literature related to transfer learning and meta-learning in the context of predictive maintenance.

Chapter 3

Input Representation and State Separation

This Chapter addresses the segmentation of vibration data gathered from roller element bearings into categories based on equipment operational states. These groups semantically represent operation modes of the equipment, such as RPM, load, external noise, etc., with the fundamental differences in equipment operation helping to cluster similar data together and apply architectures or fault detection techniques unique to the segmented groups.

This chapter is divided into two main sections; first, exploring feature representations to define the data, and second, using unsupervised learning techniques to separate the data based on those features.

Parts of this chapter have been published in the proceedings of the 27th International Congress on Sound and Vibration (ICSV27), 2021.

ABSTRACT

Reliability predictions on industrial equipment are becoming increasingly accessible as more devices become connected to sensors. Large strides in the field of predictive maintenance have been taken towards reducing downtime and optimizing operating schedules. However, extracting actionable insight from vibration data coming from rotating machinery working with variable external loads remains challenging. Due to the realities of machinery at work in real-world environments, many factors can contribute to abnormal and misleading signals: quality of the connection, infiltration of noise, and the natural task (often varying) of the machine at hand. A method is proposed that leverages vibration data to isolate the various states of a machine based on time-frequency features and an audio processing technique involving a neighbour search and local maxima. This yields a fingerprint of the signal that relies primarily on harmonic peaks, and therefore, features a robust signal to noise ratio. This fingerprint is combined with time-frequency features to form a feature vector. Feature vectors from all signal samples form a matrix, where PCA is used to reduce the data dimensions and prepare for unsupervised clustering. Clustering is done via a Gaussian mixture model. This helps define the relationship between states, and whether the data should be grouped together further. The method identifies different tasks performed by the machine and associates these tasks with a specific machine state. In the case of new tasks, the method creates a new parallel channel. By comparing the results of these isolated parallel channels, conclusions can be drawn related to machines operating under semi-variable conditions, performing different tasks that would otherwise disrupt typical vibration data sets. Tests were run on 2 different data sets and demonstrated the ability to properly separate different mechanical components mounted on the same shaft, as well as different rotational operating speeds of the machine.

3.1 Chapter Introduction

Current IFD methods are challenged by the real-world effects of noise and signal leakage from other machines in a system or work environment [56]. Moreover, changes in the machine's process or operating conditions can have a large effect on the results. Early fault warnings or prognostic efforts can be thwarted by changes in a machine's work environment due to variations in load, operating speed, or noise. This variation, often defined as multimode processes, are common in industries such as chemicals and pharmaceuticals, aviation, power generation, mining, and manufacturing [57]. This chapter looks at a new method of state extraction for use in multimode monitoring or for dealing with signal and noise leakage. Although multimode operation typically encompasses steady state, transitional, and mode variations, the current study will focus on identifying various operating modes by extracting the operating mode of the piece of equipment at any given time. For solutions involving steady state and transitional signals, further subroutines would be required for analysis.

3.2 Methodology

Image processing related to spectrogram peaks has been shown to handle noise well, especially in speech recognition and music identification applications where signal noise is common [58]. By using audio and image processing techniques, a fingerprint can be created to represent each vibration reading, and then an unsupervised classification method can be used to group each reading based on features. A recent review of clustering methods has shown that principal component analysis (PCA) dominates the unsupervised learning clustering landscape, with highlights being practicality and quick development [57]. However, Quiñones-Grueiro et al. suggest that PCA struggles with non-linear and multimode

data. Several other options for multimode classification were also investigated, including partial least squares (PLS), independent component analysis (ICA), and support vector machines (SVM). Studies have found that Gaussian mixture models (GMMs) work well even for non-Gaussian [57], non-linear, and multimode data [44]. In fact, gas turbine studies have shown that start-up and shutdown procedures can often lead to misleading conclusions [59]. In particular, these studies have explored methods for identifying these different states and accounting for variations in measurements with the use of advanced clustering methods and mixture models, many turning to GMMs [43], [44], [59].

The proposed state extraction method is described in Figure 12. It is comprised of three main parts: fingerprinting of the vibration signal, dimensional reduction, and classification of the machine's states using the GMM as a classifier. Vibration measurements from bearings in different configurations are taken using piezoelectric accelerometers.

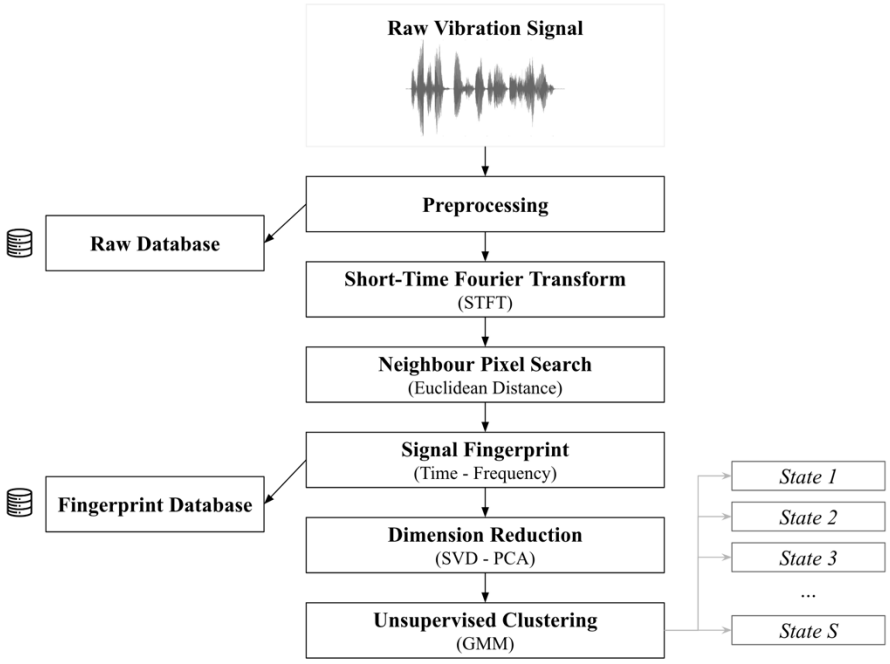


Figure 12: Overview of the proposed state extraction model

To reduce the complexity of the problem, this study focuses on machines that are cyclic in nature, having a rotational frequency of at least 10Hz (with the typical range expected to be between 10Hz and 250Hz) [60]. High-speed applications, such as gas turbines or other high-speed equipment, would need further consideration. Furthermore, vibration readings are assumed to have a sample length of at least 1s, with a sample rate of at least 20kHz. It is important to note that longer sample times would improve the quality of the output due to the larger number of continuous full rotations of the machine being included as more columns in the spectrogram. Finally, it is assumed that failure and deterioration of the components occur gradually (i.e. not within a single sample), as the algorithm is intended to be used as a precursor tool for diagnosis and prognosis.

3.2.1 Fingerprinting of the Vibration Signal

A spectrogram is used to represent the vibration signal over the entirety of the sample. It is computed using the short-time Fourier transform (STFT); effectively computing a fast Fourier transform (FFT) on a set of overlapping windows [61]. Hamming windows are chosen with a sequential overlapping of 50% (note that the longer the window overlap, the higher the resolution and, therefore, the higher the potential for capturing peaks). This is beneficial in providing better inspection of the components. However, it also increases the memory and computational requirements; the window size also affects this resolution [58]. For the purposes of this study, a window size of 512 samples was empirically determined to be optimal based on typical vibration datasets spanning 1-5 seconds in sampling time. The equation for the STFT is given as:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i k n/N}, \qquad 0 \leq n < N, \qquad 0 \leq k < N \tag{51}$$

A variety of peak finding methods can be used to derive a robust fingerprint from within the spectrogram. Common techniques consider a number of methods based on signal to noise ratio, simplicity, search speed, and discrimination power [61]. The chosen method involves creating a binary map of a local neighbourhood and then using a combination of high pass filtering and image maxima based on Euclidean distances. This technique is adapted from the work of Scholkmann et al. [62] who's algorithm automatically detected spectrogram peaks even in noisy and periodic signals. Colak et al. [63] used a similar technique to propose an automatic multiscale peak detection algorithm. The code for these works are available on Github as well as through an optimized implementation by Gotlibovych, Duarte, and Rodomansky [64], [65], for which the python code can be found in the Appendix. The work in this thesis adapted these open-source algorithms, which were originally intended for use on music for song identification. Using Python's *SciPy, SkLearn* and *NumPy* packages simplified most of the signal processing required to compute the peaks. A visualization of the peak representations after implementation is shown in Figure 13.

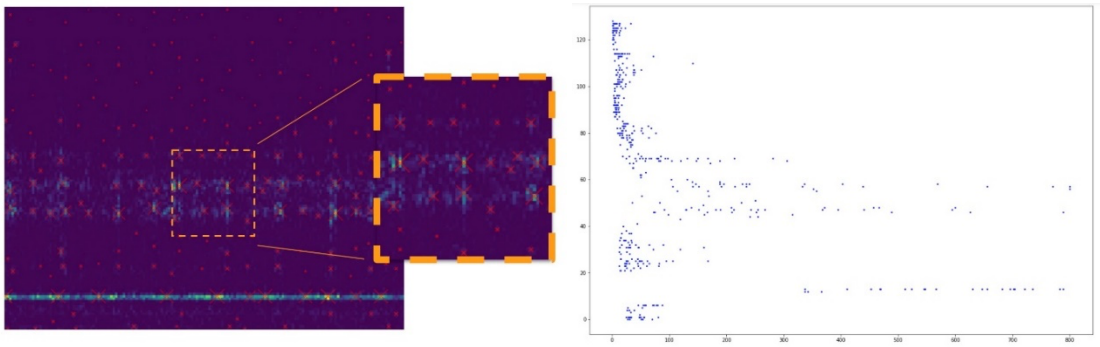


Figure 13: A visualization of the image processing peak extraction (left) and the resulting vibration fingerprint (right)

Having identified the individual time-frequency peaks (named markers), the task is then to represent the markers within the context of a fingerprint of that sample. Common audio methods form a constellation map; a set based on the location of the peaks in frequency and

time, using an anchor point [58]. However, in the interest of creating rotating machine fingerprints, the cyclical nature of the machine presents the opportunity to omit the exact time anchoring of each marker. Unlike fingerprinting used in speech or music recognition, the sequence of amplitudes and frequencies are important, but once the peaks are extracted, the sample time loses meaning if the window size is large enough to capture multiple revolutions of the machine.

3.2.2 Dimension Reduction and Gaussian Mixture Models

To simplify the process of unsupervised clustering, the fingerprints were reduced to a lower dimension while retaining as much variance as possible. This is done via singular value decomposition (SVD)-based PCA, as discussed in Section 2.4.2. The data points are projected on the axis of the eigenvector in the point cloud, which allows the dataset to be reduced in dimension while capturing as much as the variance as possible. For implementation in Python, there are multiple techniques. One could use the *numpy* library and *linalg* module to compute the covariance matrix, the eigenvectors, eigenvalues and determine the principal components. Alternatively, and perhaps more simply, one could also use the *sklearn* library to import the *PCA* module, and simply use the *pca.fit* method after initializing the principal components using the *PCA* function built into *sklearn*.

With the data reduced in dimension, GMMs function as an overlapping set of multivariate distributions. The clusters have a probability distribution, with each cluster having its own mean, variance, and size:

$$p(x) = \sum_{c} \pi_c \, \aleph(x;\, \mu_c, \sigma_c) \qquad 52$$

where the subscript c denotes the cluster number. Each of these components is described by a multivariate Gaussian distribution, given by

$$\aleph(x;\ \mu,\beta)\ =\ \frac{1}{(2\pi)^{d/2}}|\beta|^{-1/2}\ exp\left\{-\frac{1}{2}(x-\mu)^T\cdot\beta^{-1}\cdot(x-\mu)\right\} \quad 53$$

where μ is the mean, β is a matrix of variances, and d is the number of dimensions. Expectation-maximization aims to apply the appropriate number of Gaussian clusters [66]. It proceeds iteratively in two steps. In the first step, the parameters are treated as fixed for each data sample. The probability of it belonging to a given cluster component is then computed. In step two, called the maximization step, the log likelihood (the log probability of data points under the mixture model) is calculated and the data fit is increased with each iteration. A complexity penalty is used on the score (negative log likelihood of the data). This complexity penalty is modelled by the Bayesian information criterion (BIC) [67]. For the proposed algorithm, the BIC is used to locally maximize the number of starting Gaussians.

In this work, the spectrogram fingerprints are associated with the given mixture models and are updated, allowing the Gaussians to incorporate the information obtained from the new fingerprint. An output vector that contains the probability metrics of each point and how it was classified with the Gaussian is maintained. For the implementation, the *gaussianmixture* module part of the *sklearn* package was used. Figure 14 shows the Gaussian distributions, as well as the BIC optimizations. If a fingerprint shows low probability of belonging to any of the Gaussian mixtures when compared to the cumulative probabilities of the fingerprints inside that parallel, a new channel is formed and the process is restarted for that one fingerprint, now forming its new Gaussians and new parallel channel.

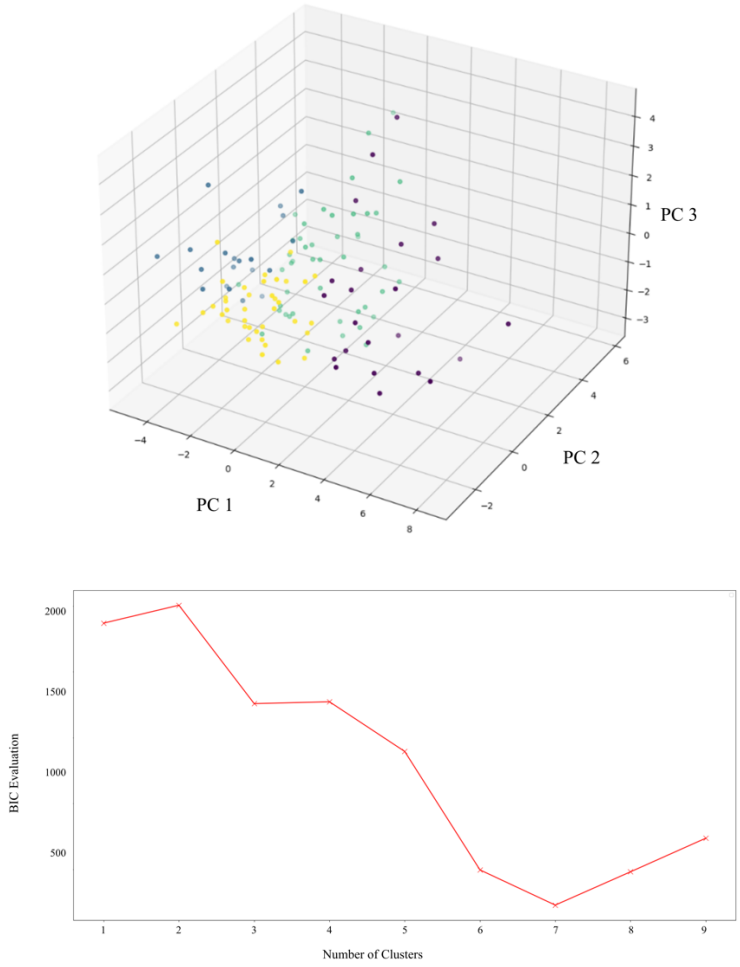


Figure 14: The Gaussian clusters (top) and the BIC optimization per number of clusters (bottom)

3.2.3 Speed Variations and Component Extraction

On a test rig shown in Figure 15, data was collected using accelerometers attached to bearings on both ends of the shaft. A healthy bearing is fixed to one end of the shaft and a faulty bearing to the other end, while a load is supported between them. 20 vibration readings were taken at incremental speeds of 600, 1200, and 1800RPM, on both the healthy and faulty bearings, giving a total of 120 readings.

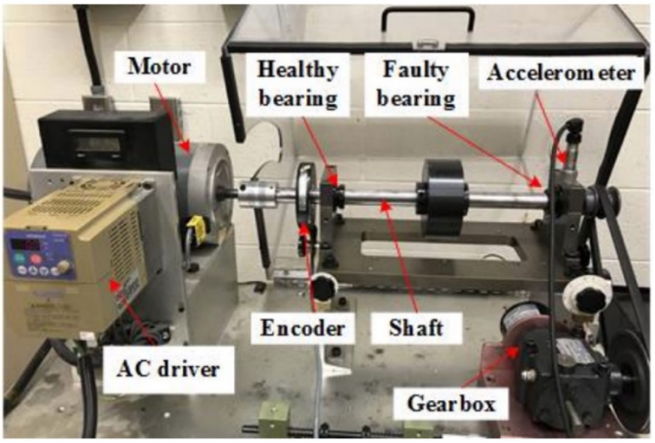


Figure 15: Test rig used to generate data, marked with various components [68]

The program was then used to classify the data into separate states, corresponding to the three operating speed increments, as well as the two different bearings. A total of 6 states were expected, with the data evenly distributed between these states. The program properly classified 107 of the readings, displaying an accuracy of 89.2%. The highest accuracy was obtained for the faulty bearings, achieving an average of 95% state extraction accuracy. Faulty bearings are thought to provide better results due to the increase in unique peaks and sidebands created by the faults at higher frequencies. The results are shown in Figure 16.

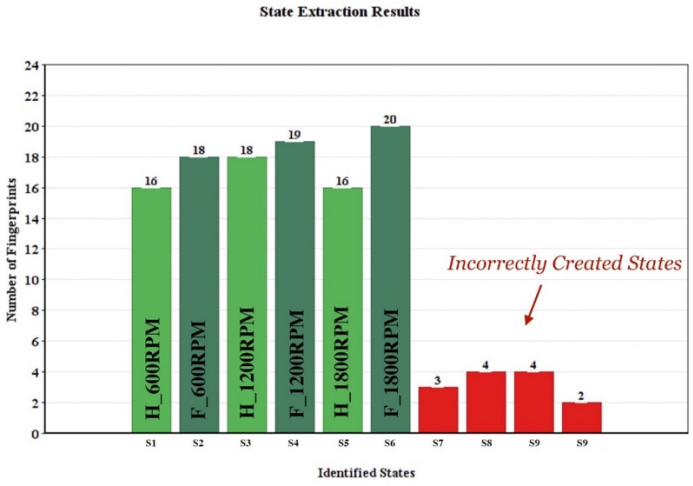


Figure 16: Test rig state classification results

Conversely, 4 incorrect states were also created due to the program's inability to properly classify the state of these readings. It is reported that these faults occurred early in the feature extraction process, implying that the program increases in state identification accuracy as more data is added to the system.

3.2.4 Accounting for Deterioration

The program must also account for the natural changes in the vibration signal generated by a given component due to its deterioration over time. Vibration data from the NASA run-to-failure experiments expose bearings to severe deterioration [69]. Four bearings were loaded with 6000lbs and were run at a constant speed of 2000RPM over three experiments. Experiments spanned timeframes of 1 week for dataset 2, and about a month for dataset 1 and 3. These datasets present a good trial for GMM classification by accounting for bearing deterioration. The classification and further grouping of vibration fingerprints can be seen in Figure 17.

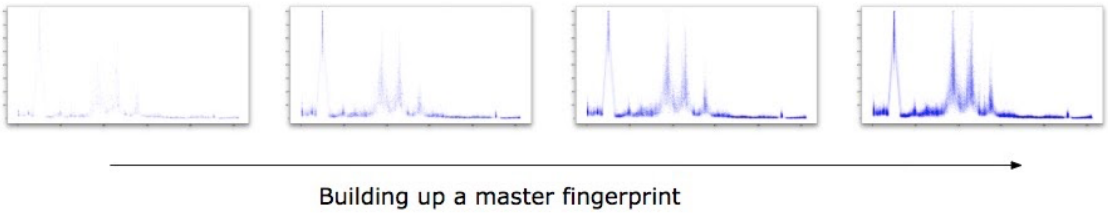


Figure 17: The final superimposed fingerprints from one of the NASA vibration datasets

Based on the results of the 4 bearings that were tested, an average of 97% of the data was recognized and classified as a segmentation. Certain samples were improperly classified as new states, as they possessed outliers and other features significantly unlike the rest of the data. Future work should consider more robust methods of preprocessing and outlier detection to improve accuracy.

3.3 Chapter Discussion and Conclusions

The fingerprinting technique proposed in this paper has been shown to be an effective method for capturing relevant data, while simultaneously reducing the number of points required to carry along. The window sizing must be configured to avoid an excessive number of captured peaks. In some cases, despite little spectral activity, the program still associates peaks because of the use of nearest neighbour comparisons. Therefore, a threshold given as a ratio of the root mean square (RMS) value of the signal is introduced. These peaks can still be kept in the main fingerprint, but are left out of the GMM process.

When testing the program on the lab obtained data generated at different speed increments for two different bearings, initial values for the number of clusters posed a problem. This resulted in a drop in early accuracy of the program, and led to new states being falsely created for the vibration fingerprints that couldn't be associated with existing states. A more efficient method for early cluster initialization is required. This error would likely propagate dramatically for cases involving an increased number of expected states. Efforts have been made to improve this decision process, including the use of kernel density estimation, the Bayesian Ying-Yang criterion, and entropy operators [57] [70].

In this work, a novel state extraction algorithm is proposed which uses image and audio processing techniques to generate a 2D fingerprint of a vibration sample. A Gaussian mixture model is then used to associate these fingerprints with correct states and modes of the machine. The program was shown to be 97% accurate in classifying stationary deterioration data and 89.2% accurate in classifying multi-mode data provided by two separate bearings. Improvements to the program could be made in choosing the initial clusters to represent the fingerprints.

Chapter 4

CNN Architecture and Generalization

This chapter addresses the challenge of generalizing a single deep learning network to perform intelligent fault detection on a wider range of equipment scenarios, such as different equipment types, sizes, operating speeds, and fault severities. Fault detection is still performed on roller elements within these different pieces of equipment, but the network is designed to adapt to the various circumstance acting on the bearing. The goal of this chapter is to uncover specific factors that either promote or prevent generalization, and to use these to the network's advantage in delivering classification results over a wider range of equipment.

Parts of this chapter have been published in the proceedings of the ASME 2021 International Design Engineering Technical Conference and Computers and Information in Engineering Conference (EDETC/CIE2021).

ABSTRACT

Through the intelligent classification of bearing faults, predictive maintenance provides for the possibility of service schedule, inventory, maintenance, and safety optimization. However, real-world rotating machinery undergoes a variety of operating conditions, fault conditions, and noise. Due to these factors, it is often required that fault detection algorithms perform accurately on data outside its trained domain. Although open-source datasets offer an opportunity to advance the performance of predictive maintenance technology and methods, more research is required to develop algorithms capable of generalized intelligent fault detection across domains and discrepancies. In this study, current benchmarks on source–target domain discrepancy challenges are reviewed using the Case Western Reserve University (CWRU) and the Paderborn University (PbU) datasets. A convolutional neural network (CNN) architecture and data augmentation technique more suitable for generalization tasks is proposed and tested against existing benchmarks on the PbU dataset by training on artificial faults and testing on real faults. The proposed method improves fault classification by 13.35%, with less than half the standard deviation of the compared benchmark. Transfer learning is then used to leverage the larger PbU dataset to make predictions on the CWRU dataset under a challenging source-target domain discrepancy in which there is minimal training data to adequately represent unseen bearing faults. The transfer learning-based CNN is found to be capable of generalizing across two open-source datasets, resulting in an improvement in accuracy from 53.1% to 68.3%.

4.1 Chapter Introduction

Neural networks and traditional machine learning classifiers improve IFD performance over signal processing and standard mathematical solutions, but criticism of these models typically involves their reliance on the ability of the extracted features to capture fault information, which often suffer from the non-linear realities of time-domain signals [10].

Deep learning methods present an exciting opportunity to push the frontier of what is possible for intelligent fault detection. Layers of deep networks consist of non-linear activation functions, with an incredible capacity for function approximation [71]. The stacking of these layers allows abstraction and generalization of non-linear features, which results in a framework perfectly suited for solving complex problems. Typical approaches for deep learning based IFD involve the extraction of time domain or frequency domain features from a vibration dataset, which are then supplied as inputs to a deep learning algorithm such as a CNN [72]. Raw data inputs are also sometimes fed into a deep learning network or even, in some cases, into a 1D CNN [30]. Deep networks shift the need for expertise related to feature engineering towards the need for large quantities of quality data. In most predictive maintenance use cases, having proper distributions of labeled data is a challenging task. Since machinery faults or breakdowns are rare and undesirable, gathering fault data is uncommon. Furthermore, changes in operating conditions (RPM, load, force), fault conditions (type, location, severity) and noise (data connectivity, nearby equipment, other processes) result in a large variation in the data being collected and analyzed. Algorithms trained on specific tasks, but capable of generalizing to new tasks and to discrepancies between source and target data domains are a key asset in both academia and industry.

Recently, the adaptive CNN (ADCIN) has been used to improve training speed and accuracy [73], and the deep convolutional transfer learning network (DCTLN) introduces a

combination of two modules: first to learn the features and recognize health conditions using a 1D CNN, and then a domain adaptation module to minimize the probability distribution distance while learning features [74]. This method is unique in the way that it operates without the need for labels in the target domain. Three datasets are used for interchangeable transfers, leading to 6 results with a strong performance of 86% accuracy on average across the target datasets. Zhang proposes a deep convolutional neural network with wide first-layer kernels (WDCNN), where wide first layer kernels are used as a method for processing more features and improving robustness to noise [75]. In this case, data augmentation techniques are also used, along with the method, to achieve a 92% average accuracy across operational conditions. However, there is still very little research that has explored factors in convolutional networks that improve generalization performance, which is the goal of this chapter.

4.1.1 Data Description

4.1.1.1 CWRU Bearing Fault Dataset

The Case Western Reserve University (CWRU) dataset, consisting of an electric motor, torque transducer, and dynamometer, shown in Figure 18, is a popular dataset in the bearing vibration literature [34].

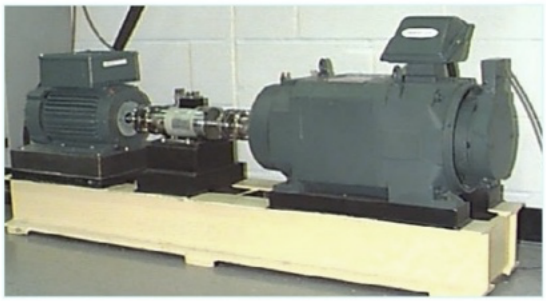


Figure 18: The Case Western Reserve University (CWRU) test rig [34].

Four operating conditions are recorded by varying the motor speed between 1797RPM, 1772RPM, 1750RPM, and 1730RPM. It is important to note that speed variations are minimal when used to define training and testing data, indicating that there is a lack of training and testing data diversity for domain adaptation across speeds. The dataset also includes readings from four different bearing health states (Baseline, Inner Race (IR), Outer Race (OR), and Ball). The outer race condition features three test positions (centered, orthogonal, and opposite to the load). There is data for both drive end and fan end bearings. Seeded faults are created using electro-discharge machining (EDM) with fault sizes of 0.007, 0.014, 0.021, and 0.028 inches. 10 second samples were collected at 12kHz and 48kHz. Figure 19 provides detailed information about the dataset.

Fault Diameter	Motor Load (HP)	Motor Speed (rpm)	Inner Race Measurements	Ball Measurements	Outer Race Measurements
0.007"	0	1797	IR007_0	B007_0	OR007@6_0
	1	1772	IR007_1	B007_1	OR007@6_1
	2	1750	IR007_2	B007_2	OR007@6_2
	3	1730	IR007_3	B007_3	OR007@6_3
0.014"	0	1797	IR014_0	B014_0	OR014@6_0
	1	1772	IR014_1	B014_1	OR014@6_1
	2	1750	IR014_2	B014_2	OR014@6_2
	3	1730	IR014_3	B014_3	OR014@6_3
0.021"	0	1797	IR021_0	B021_0	OR021@6_0
	1	1772	IR021_1	B021_1	OR021@6_1
	2	1750	IR021_2	B021_2	OR021@6_2
	3	1730	IR021_3	B021_3	OR021@6_3

Rotational Speed (RPM)	Load (HP)	Damage Size (in)	Name
1772, 1750, 1730	1,2,3	0.007	A
1772, 1750, 1730	1,2,3	0.014	B
1772, 1750, 1730	1,2,3	0.021	C

Figure 19: CWRU Dataset description (top table describing file names and conditions and the bottom table describing the speed, load, and faults of the datasets)

4.1.1.2 Paderborn Bearing Dataset

The Paderborn University (PbU) dataset consists of an electric motor, load motor, flywheel, torque measurement shaft, and rolling bearing test module, shown in Figure 20 [76]. One of the unique advantages of the Paderborn dataset is that data is collected for both artificially seeded faults, as well as real faults caused by accelerated life testing. This unique feature of the PbU dataset provides the opportunity to test algorithms across these boundaries.

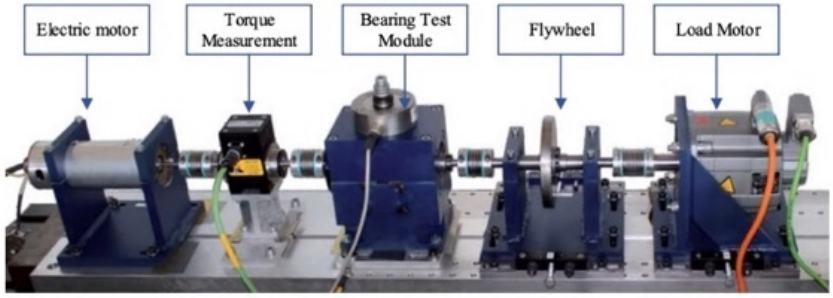


Figure 20: The Paderborn University (PbU) test rig [76].

Four different conditions are recorded, with varying operating parameters, including rotational speed (900 and 1500RPM), torque (0.1 and 0.7Nm) and radial force (400 and 1000N). Run-in periods are given for healthy bearings to track any fault growth. In terms of artificial faults, inner race faults and outer race faults are seeded with varying degrees of damage, as shown in Figure 21, using methods such as EDM, electric engraving (EM), and drilling. Data collection was performed at 64kHz, with 20 measurements of 4 second samples.

Bearing Code	Component	Extent of Damage (level)	Damage Method
KA01	OR	1	EDM
KA03	OR	2	electric engraver
KA05	OR	1	electric engraver
KA06	OR	2	electric engraver
KA07	OR	1	drilling
KA08	OR	2	drilling
KA09	OR	2	drilling
KI01	IR	1	EDM
KI03	IR	1	electric engraver
KI05	IR	1	electric engraver
KI07	IR	2	electric engraver
KI08	IR	2	electric engraver
OR: outer ring; IR: inner ring;			

(a)

Class		Training	Testing
1	Healthy	K002	K001
2	OR Damage		KA22
		KA01	KA04
		KA05	KA15
		KA07	KA30
			KA16
3	IR Damage		KI14
		KI01	KI21
		KI05	KI17
		KI07	KI18
			KI16

(c)

Bearing code	Damage (main mode and symptom)	Bearing element	Combination	Arrangement	Extent of damage	Characteristic of damage
KA04	fatigue: pitting	OR	S	no repetition	1	single point
KA15	Plastic deform.: Indentations	OR	S	no repetition	1	single point
KA16	fatigue: pitting	OR	R	random	2	single point
KA22	fatigue: pitting	OR	S	no repetition	1	single point
KA30	Plastic deform.: Indentations	OR	R	random	1	distributed
KB23	fatigue: pitting	IR (+OR)	M	random	2	single point
KB24	fatigue: pitting	IR (+OR)	M	no repetition	3	distributed
KB27	Plastic deform.: Indentations	OR + IR	M	random	1	distributed
KI04	fatigue: pitting	IR	M	no repetition	1	single point
KI14	fatigue: pitting	IR	M	no repetition	1	single point
KI16	fatigue: pitting	IR	S	no repetition	3	single point
KI17	fatigue: pitting	IR	R	random	1	single point
KI18	fatigue: pitting	IR	S	no repetition	2	single point
KI21	fatigue: pitting	IR	S	no repetition	1	single point
OR: outer ring; IR: inner ring; S: single damage; R: repetitive damage; M: multiple damage						

(b)

Figure 21: PbU dataset description [76]

4.1.2 Dataset Similarities and Opportunities for Source-Target Discrepancy Tasks

Both the Case Western and Paderborn datasets offer the potential to test deep learning models geared towards generalization. For Paderborn, there is the possibility to transfer from artificial to real faults, where real fault data consists of features outside the scope of the artificial data samples. For Case Western, the same opportunity presents itself, but for fault severity instead, where unseen bearing conditions are split based on fault size. Both datasets also provide multiple operating conditions, increasing the level of feature diversity. By re-

arranging the dataset parameters, as in Table 6, similar conditions for both datasets can be identified.

Table 6: Combined datasets of CWRU and PbU in relation to each other

		Operating Condition			Fault Condition	
Dataset	Component	Rotational Speed (RPM)	Load (HP)	Force (N)	Extent of Damage (in)	Damage Method
Case Western*	Baseline	1730, 1750, 1772	1,2,3 HP	-	-	-
	Inner Race	1730, 1750, 1772	1,2,3 HP	-	0.007, 0.014, 0.021	EDM
	Outer Race	1730, 1750, 1772	1,2,3 HP	-	0.007, 0.014, 0.022	EDM
	Ball	1730, 1750, 1772	1,2,3 HP	-	0.007, 0.014, 0.023	EDM
Paderborn**	Baseline	1500		1000 - 3000	-	-
	Inner Race	1500	-	400, 1000	< 0.079 | 0.079 - 0.18	EDM, EG
	Outer Race	1500	-	400, 1000	< 0.079 | 0.079 - 0.18	EDM, EG, Drilling

* Not including 0 HP operating mode, and 2 of 3 outer race positions
** Not including 900 RPM operating mode

Three of the four PbU operating conditions are similar in speed (1500RPM) to the CWRU dataset (1730-1797 RPM). Furthermore, both datasets feature artificial damage to both inner and outer races caused by EDM. The similarity between datasets can be emphasized by grouping parameters under *Operating Conditions* and *Fault Conditions*. This organization of data is also a realistic way that data would be organized in industry, as data coming from each machine could be organized by searchable tags. By deploying a similar strategy, a data model could be created that would allow the user to search the database for operating condition and fault condition tags. This observed similarity between datasets also serves as a precursor to potentially combining these datasets into a larger and deeper architecture capable of generalization.

4.1.3 Problem Description

Most CNN algorithms found in the literature that use the CWRU dataset have a train/test split across different motor loads (HP), providing average accuracy results above 85% [72]. However, when the train/test split is modified to occur across fault sizes, average accuracy drops significantly to less than 55% [77]. Interestingly, this second case presents a more

realistic domain adaptation challenge, where known faults are used to train algorithms that can identify unseen faults of different severities. A spectrogram visualization of these two data acquisition regimes are shown in Figure 22. The difference between these two train/test splits is visualized by comparing the splits by motor load (by HP - most common method) to that of fault severity (proposed method). Figure 22 also elaborates on the PbU dataset, in which a challenging discrepancy exists between artificial faults and real faults of the same type and severity, collected under similar operating conditions. Due to these data discrepancies, fault prediction results on domain adaptation tasks for both the CWRU and PbU datasets are generally much lower [78].

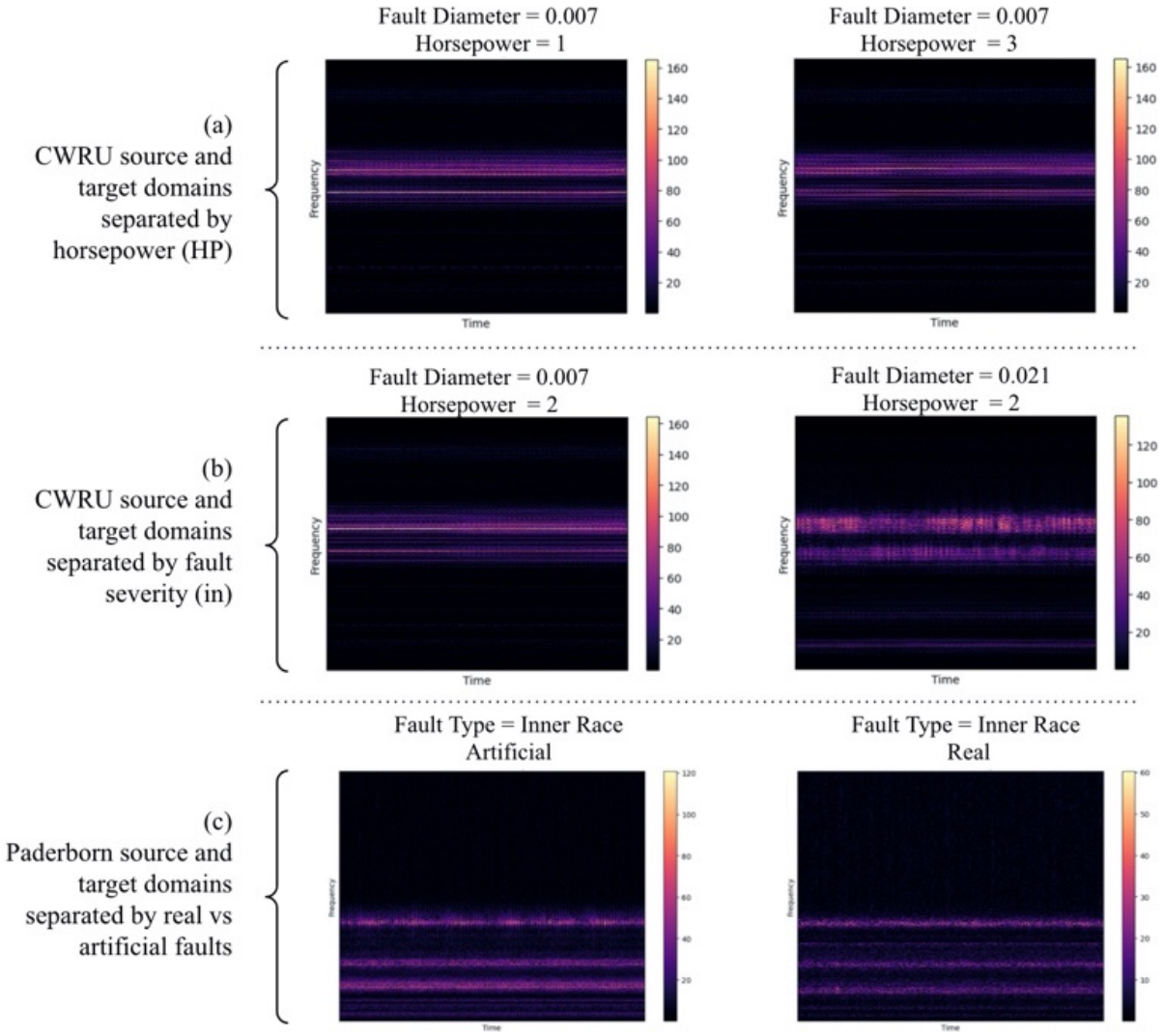


Figure 22: It can be seen that minimal discrepancy exists between the two defined operating states in (a), minimizing the need for domain adaptation. In case (b), the opposite is true; there is a large difference between 0.007 and 0.021 faults. Case (c) is the difference between artificial and real faults and is similar to case (b).

4.1.4 Solution Description

Having defined the discrepancies between source and target domain data as an important problem to address, this study will improve the current obtainable accuracy of the CWRU severity dataset split by leveraging transfer learning from a larger more generalized model trained on the PbU dataset.

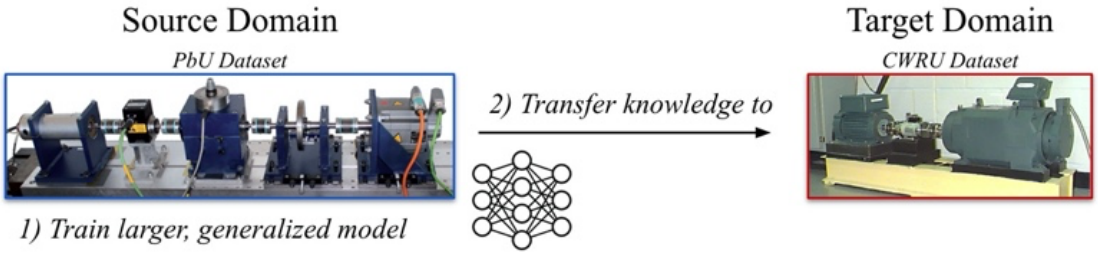


Figure 23: Solution description

Data augmentation techniques will also be incorporated to further improve performance of the network. This combination of two open-source datasets to improve accuracy on challenging target domains will pave the way for larger generalized models and an infrastructure capable of adapting to unseen faults and applications.

4.2 Chapter Methodology

4.2.1 CNN Architecture Modifications to Improve Generalization

Under the usual considerations of a CNN, the kernels typically act as feature detectors, identifying lines and edges in an image. These features are then combined by subsequent layers to form shapes, and eventually objects. This hierarchical combination is the product of a stack of small filters working to form larger more complex shapes, where intermediate layers of the network act as levels of abstraction. Intuitively, this means that deeper models should improve generalization, whereas wide models may improve memorization via the introduction of non-linearities [78]. Despite the ongoing debate in this area, it is possible to use this intuition to help guide the development of a new model. Evidently, balancing the

depth (number of layers) and width (number of filters) of a given model with the appropriate number of hyperparameters is key to managing the trade-off between overfitting and generalization. Specific strategies can also be used to aid in the prevention of overfitting. Parameter norm penalties, such as L1 and L2 regularization, add an extra term to the weight update function in an effort to improve a model's robustness against outliers and noise by regulating the growth of certain weight functions [63]. Batch normalization can be used to standardize the layer inputs for each minibatch processed through the algorithm. Early stopping can be used to prematurely halt the algorithm before the validation error begins to rise (indicating that the model may be starting to memorize features rather than improving generalization) [79]. One drawback of this method is that the model may not be making use of all available data. Therefore, relying on this technique should be avoided, and closely monitoring training and validation errors is necessary. When trying to avoid overfitting, reducing the complexity of the model should be incentivized and smaller and simpler models with fewer hyperparameters should be prioritized.

By drawing inspiration from the WDCNN method, which uses wide first layer kernels [75], new models for this program were tested by stacking smaller square filters in order to increase the receptive field, as seen in Figure 10. The stacking of smaller square filters results in fewer parameters when compared to directly imputing large filters. Furthermore, greater non-linearities are packed between them, leading to a more expressive feature representation and more powerful features at the cost of more memory [80]. However, seeing as the goal is to develop generalized transfer learning across domains, training time is reduced when compared to training new models from scratch, justifying the sacrifice in memory. As the receptive field is grown, the depth of the model is managed by introducing bottlenecks of 1x1 depth-wise convolutions (shown in the bottom of Figure 24) that constrict the network, where these lower dimension layers are added to regularize the training and aid in the prevention

of overfitting [81]. For implementation in Python, the *Keras* library was used along with *Tensorflow*.

This structure allows for a set of broadened introductory layers with more hyperparameters in the early stage triple convolutional layers, followed by a forced constriction to encourage more generalization of the features learnt. The network architecture of the model used for objective 1 is shown in Figure 24 and Table 7.

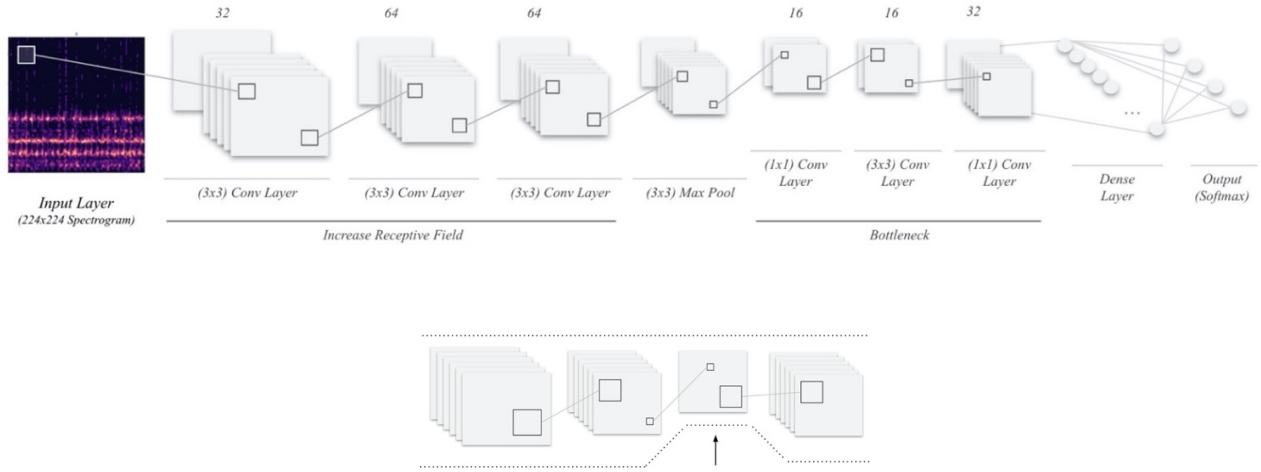


Figure 24: CNN architecture designed for domain adaptation tasks (top) and the bottleneck layers' effect on the width of a convolutional neural network (bottom).

Table 7: CNN Architecture applied to the PbU dataset

Layer	Depth	Kernel Size	Stride	Activation	Regularizer
Input					
2D Convolution	32	3,3	2,2	ReLu	L2
2D Convolution	64	3,3	2,2	ReLu	L2
2D Convolution	64	3,3	2,2	ReLu	L2
2D MaxPool		3,3	2,2	-	-
Batch Normalization	-	-	-	-	-
2D Convolution	16	1,1	1,1	ReLu	L2
2D Convolution	16	3,3	2,2	ReLu	L2
2D Convolution	32	1,1	1,1	ReLu	L2
	Fully Connected Layer				
Dense (Dropout = 0.4)	16	-	-	ReLu	L2
Dense	4	-	-	Softmax	

Batch-size: 32, number of epochs: 40, Learning rate: 0.001 (Step Decay)

4.2.2 Data Augmentation Techniques to Reduce Source and Target Domain Discrepancy

Data augmentation is applied to minimize the discrepancy between the source and target domain. A spectrogram is used as input by applying a short time Fourier transform (given by equation 51), effectively sliding a window containing the fast Fourier algorithm across the time series signal [61]. On this spectrogram, bearing faults may manifest themselves as peaks or sidebands of the fault frequencies. These signal patterns have meaning not only in the value of their magnitude, but in their location on the pitch and temporal scale. Horizontal and vertical flips are applied to random samples in the training and target datasets to aid the model in generalizing fault behavior into features that avoid fixation on a specific spectrogram location. To smooth out and reduce the impact of changes in operational speed, random speed scaling and pitch shifting is applied, changing the nature of the fundamental frequencies at play [82]. The operating frequency is the one variable of the characteristic roller element fault equations that cannot be cancelled out, so it is important to address this. Finally, random noise is applied to the data to reduce overfitting by preventing the model from memorizing patterns. These data augmentation techniques are shown in Figure 25 and are applied at random by computing a 50% chance on each sample for each discrete augmentation.

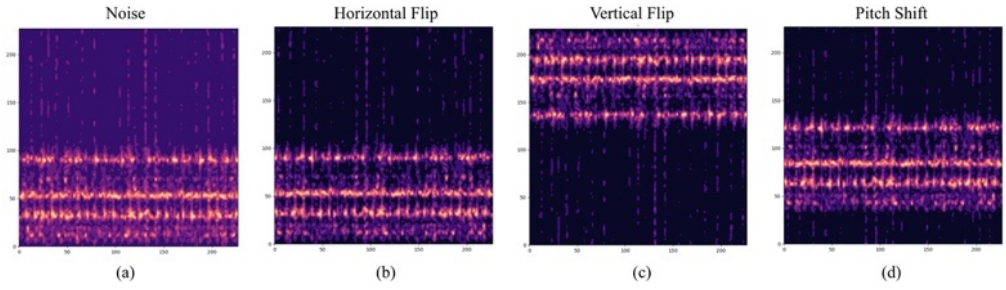


Figure 25: Data augmentation techniques applied to the spectrograms

4.2.3 Transfer Learning

Transfer learning can be described as the re-use of a model that has been trained on a given task A to help solve a given task B, where tasks A and B are similar, but with different data distributions [74]. Transfer learning presents the unique opportunity to bridge the gap between new applications in predictive maintenance, providing the possibility for successful source-target domain leaps and generalization to unseen circumstances. Studies have shown that pre-trained models such as ImageNet and ResNet, designed for image classification tasks, can be tuned to make accurate predictions using spectrograms [78]. This lends weight to the idea that spectrograms are in fact an appropriate and useful representation of the data as inputs to a deep CNN. To better explore the nature of the problem at hand, multiple CNN architectures will be purpose built for this study.

4.2.4 Experimental Setup and Process

For this study, all data was preprocessed by resampling at 48kHz (note that no resampling was required for CWRU data, but resampling of PbU data reduced the number of data points per sample from ~256,000 to ~200,000). Outlier detection and zero mean was then applied. The STFT was computed using a window size of 256 and step size of 128, yielding an output shape of (129, 1566). The spectrogram output was then cut into equal length segments of 129 by 129, resulting in 24 segments per sample with overlap considered. Data augmentation was then applied, with each transformation having a 50% probability of taking effect. The spectrograms were then normalized, labelled, and stored in a database. Figure 26 shows the data preprocessing, segmentation, augmentation, and labelling process.

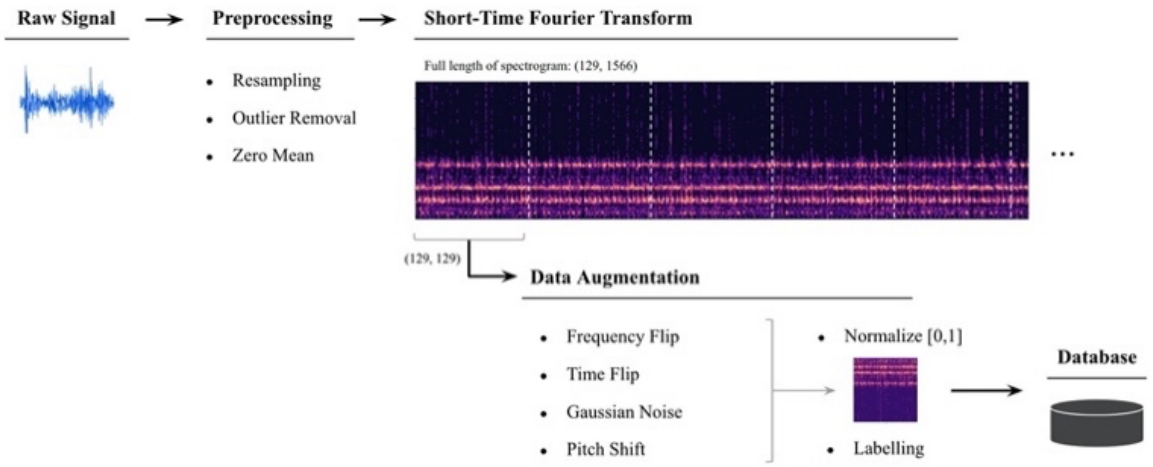


Figure 26: Process for automatic data augmentation

The CNN architectures, hyperparameters, and performances were tracked using Google's Tensorboard, using the hyperparameters as the model's naming scheme (e.g. 64-depth-3x3-kernel-3-ConvLayer-8-FClayer-05-Dropout-...). This allowed for quick searches, interpretation, and iteration of the results. Since multiple architectures and datasets were used to train and test algorithms for this study, K80 GPUs were used to reduce model training time, and models were run overnight or at off-peak hours.

4.3 Chapter Results and Discussion

Because the PbU dataset features artificial and real fault data, the discrepancy between these two datasets was used to build a CNN model capable of generalization. The process of designing this model was to run loops of the machine learning training algorithm, with sequential and small modifications to the hyperparameters after each loop. Close attention was paid to the validation error. Figure 27 shows a reduction in validation error for early epochs. However, after 10 epochs, the error on most models begins to grow. This indicates that the model has stopped generalizing and is beginning to memorize the training set. Because this study deals with discrepancy between source and target domains, this

memorization leads to obvious increases in validation error. Widening the receptive field was introduced by stacking multiple convolution layers without any pooling layers. This had an immediate positive impact on the model. Original inspiration for this was drawn from Zhang et al. [75], in which wide first layer kernels were implemented on 1D signals. Larger kernels resulted in poorer performance on average. An increase in first kernel size to 5x5 or 7x7 resulted in up to a 10% drop in accuracy and 25% increase in validation error. By stacking a series of small kernels, without introducing downsampling, through max pooling layers, the receptive field was grown without needing to use large kernel sizes.

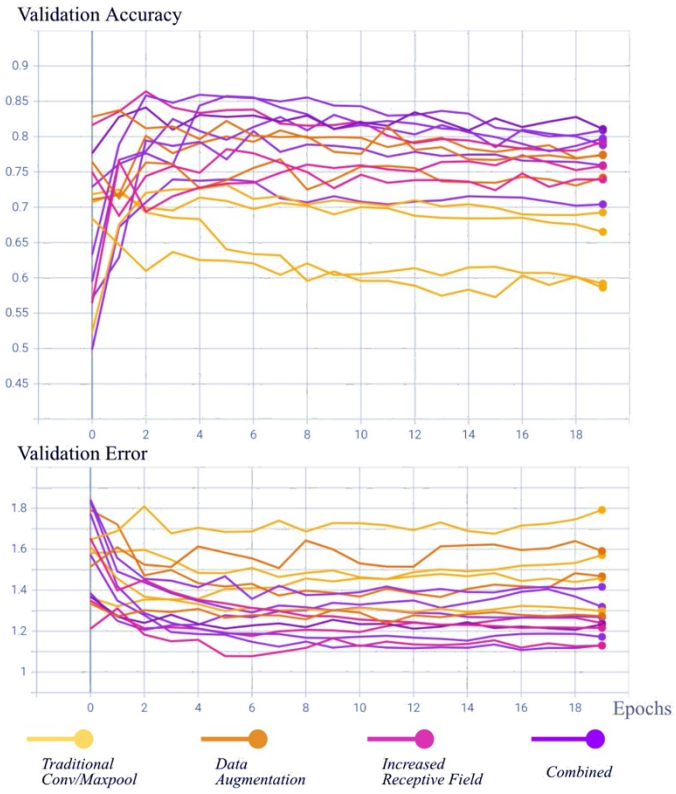


Figure 27: Validation accuracy and error for a CNN trained on the PbU seeded fault data and tested on real data.

Intuitively, since stationary signals are used in this study, larger strides along the temporal axis were tested and corresponding accuracies measured. Strides of up to the length of the kernel itself could be taken along the temporal axis without diminishing returns in

accuracy. However, this resulted in greater standard deviations for the resulting performance tests, and the technique was later discarded. An important reflection when using spectrograms as inputs to a CNN is to consider the lack of semantic context when compared to a regular image. In this case, the spectrogram inputs have vibration signatures that do not necessarily belong to a single source, as these signatures may be a product of multiple accumulated signals or interactions. This is in stark contrast to having pixels typically refer specifically to a given object in field. Rather than spatial context, it is within time and frequency components that a feature may present itself. Moreover, CNNs are considered to follow a bag of features model, in which low-level features are hierarchically combined to form larger more complex ones, but not necessarily in a comprehensive manner. In fact, the specific location of those features may not be relevant to the CNN as much as their mere presence in the image [83]. This poses some concern for spectrogram-based applications, in which both the presence of signals such as sidebands and peaks must be recognized, as well as their position relative to other signals in the spectrum. Although counterintuitive, vertical and horizontal flips did show improvement in generalization performance, perhaps addressing this issue. Nanni et al. also used rotation of the spectrogram along with horizontal and vertical flips [82]. However, the addition of this feature did not result in any gains for this study.

The final results of the model trained on PbU seeded fault data and tested on real data are provided in Table 5. Benchmarks are provided by Pandhare et. al with respect to classification accuracy on Paderborn bearings when training on artificial faults and testing on real faults [84]. The benchmarked models include k-nearest neighbor (kNN), support vector machines (SVM), time-domain CNN (TDCNN), and spectral CNN (SCNN).

Table 8: Validation accuracies for training on artificial data and testing on real data

Method	Mean Accuracy (%)
TDCNN	61.9
SCNN	59.3
SVM	53.0
kNN	52.3
Proposed Method	**75.2**

This optimized model was then ready to be transferred to the task of identifying bearing faults in the CWRU dataset. All layers, other than the fully connected layers, were transferred, with the first three convolutions frozen from any further training. The learning rate was lowered to 0.0001, and exponential decay was used for fine tuning.

Results are provided for training on dataset A and testing on dataset B according to the data groupings in Figure 19 (bottom). The model was run 5 times, with the average accuracy being 78.5% with a standard deviation +/- 6.5%. Confusion matrices for two runs, with and without transfer learning, are plotted in Figure 28. The top two matrices feature a similar convolutional neural network without transfer learning. It is observed that the main source of inaccuracy is the misclassification of the fault states. Healthy state identification is consistent with perfect or near perfect performance. However, the algorithm fails to distinguish between inner race and outer race faults.

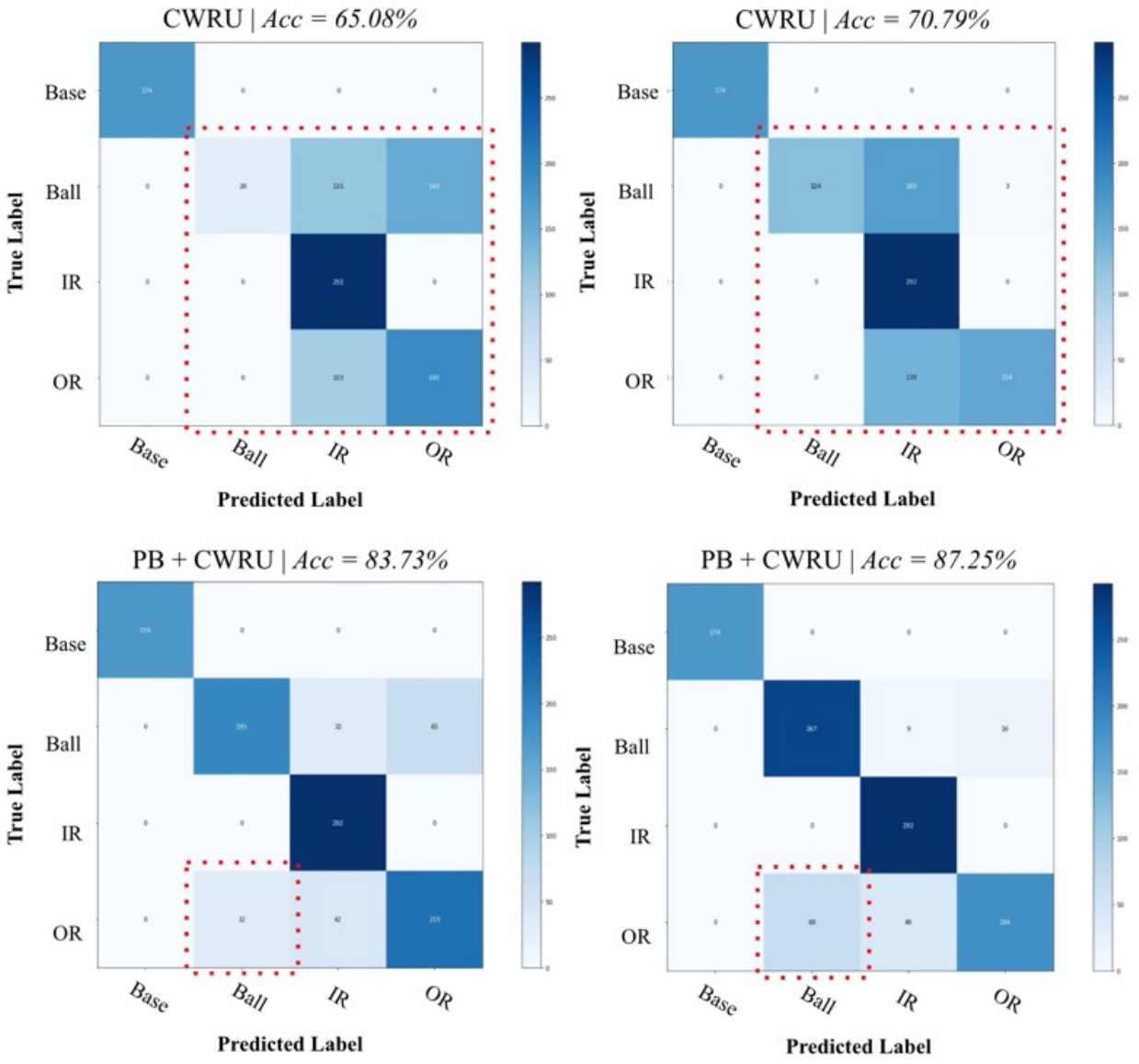
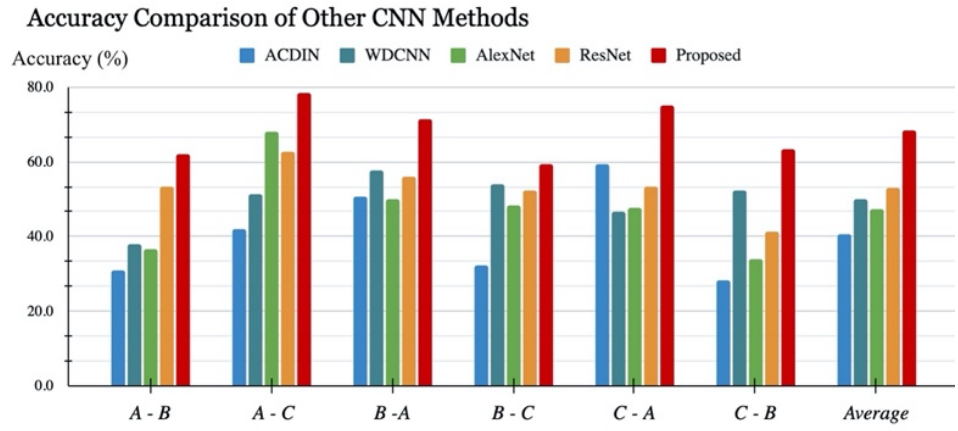


Figure 28: Confusion matrices for classification accuracy for two cases without transfer learning (top) and two cases with transfer learning (bottom). Areas of interest are highlighted in red.

A potential solution would be to simply reduce the solution space to 2 classes: healthy and unhealthy. From an operational standpoint, knowledge of the specific bearing fault is often unnecessary, as entire bearings are typically replaced either way. However, this would introduce a large class imbalance, as most realistic scenarios have very limited faulty data

to begin with. Having a biased ratio of healthy versus faulty bearing data, and a very small statistical distribution that diversifies this healthy dataset would lead most algorithms to memorize baseline conditions. In fact, almost any architecture can identify the healthy versus faulty state with 100% accuracy, as shown by the top two confusion matrices for the case without transfer learning. However, with the addition of transfer learning, generalizations from the PbU inner race and outer race fault states can be transferred to the CWRU dataset, with the benefit of sharing knowledge learned in one domain to another. This results in a significant improvement for the classification accuracy between these faults. Unfortunately, it can be observed that the transfer learning approach also introduces inaccuracies in the ball fault classifications. This is logical, as the source domain the algorithm was trained on did not have ball faults (the PbU dataset only contains baseline, inner race, and outer race fault conditions). The final accuracies obtained for this transfer learning approach, compared with other approaches, are shown in Figure 29.



	A - B	A - C	B -A	B - C	C - A	C - B	Average
ACDIN	31.0	41.9	50.8	32.2	59.3	28.2	40.6
WDCNN	38.1	51.2	57.7	54.1	46.8	52.2	50.0
AlexNet	36.5	68.0	50.0	48.2	47.7	33.8	47.4
ResNet	53.2	62.6	55.9	52.3	53.2	41.4	53.1
Proposed	**62.1**	**78.5**	**71.5**	**59.2**	**75.1**	**63.3**	**68.3**

Figure 29: Testing accuracy for Case Western when splitting the dataset by fault severity. The two letters under each set of bars correspond to the training dataset – testing dataset respectively, according to the data groupings in Figure 19 (bottom).

Improvements across all tested scenarios can be seen, with the highest average accuracy being achieved when training on dataset A and testing on dataset B. It is evident that the new method of splitting training and testing data yields a more challenging task for the network when classifying appropriate faults. The question still remains whether the improvement in performance is due to the architecture or the analysis in transfer learning from a previous domain. Other studies in the audio classification field have shown that larger pre-trained models, such as DenseNet, tend to perform well on spectrogram-based classification, even when pre-trained on ImageNet datasets [85]. This lends weight to the theory that transfer learning methodologies remain functional even under large data discrepancies provided there is a substantially large amount of data. Further work should investigate these effects by benchmarking various pre-trained architectures to identify commonalities or outliers in their performances.

4.4 Chapter Conclusion

The transfer learning method developed in this study demonstrates strong performance using a lightweight model capable of generalization across two datasets. A model architecture which prioritizes generalization by expanding the receptive field with filter stacking and then bottlenecking the model width is developed in conjunction with data analysis from two distinct datasets. The result is an increase in performance when compared to models trained on one dataset alone. However, specific error classes are introduced. Future work should group larger datasets with more fault data to avoid errors. The proposed method should also be applied to a third dataset to verify global generalization.

Chapter 5

Transfer Learning and Ensemble Model Output

This Chapter takes the concepts in Chapters 3 and 4, ties them together, and works towards building a consolidated final program to automatic as much of the machine learning pipeline as possible. First, the concepts of feature representation and state separation as described in Chapter 3 are elaborated to develop a data augmentation and task separation module. Then, using concepts explored in Chapter 4, a system is developed to generalize and guide the architecting and training of 2D convolutional networks applied to vibration spectrograms, as well as MFCC-based neural networks. Finally, a distance measure is introduced to guide the automation of model building and training, and an ensemble model is used to compute the final output. The result is an end-to-end program which takes in raw vibration data and outputs a classification prediction for the presence of a fault in a bearing, without the need for a human in the loop for model building or training.

Parts of this chapter have been published in the proceedings of the 2021 IEEE International Symposium on Robotic and Sensors Environments (ROSE).

ABSTRACT

The act of leveraging sensor data, such as accelerometers on connected industrial equipment, to diagnose and predict failures in roller element bearings is of growing interest in industrial and commercial applications. To do this, a large effort must be made to gather and process data, then apply and manage algorithms to make predictions and derive value. For these reasons, knowledge from previous models and tools to help in the evaluation and adaptation of new data sources using methods such as transfer learning and meta-learning have received significant attention in recent years. This study aims to provide a methodology for combining such tools and presents a framework for an automated machine learning (AutoML) process by using a combination of pre-existing AutoML tools and new transfer learning methodologies. The framework uses recent observations in vibration analysis and deep learning architectures to abstract away many of the small, but intricate, decisions required for data preparation and model building. This allows for a unique data science strategy and dramatic reduction in the development time required to achieve competitive baseline models when compared to industry averages, while simultaneously improving the ability to address unseen environments. First, a signal processing engine is used to standardize the data, then fingerprinting and unsupervised learning is used to separate data into groups based on a machine's operating state. Meta-learning is then used to evaluate these states as tasks to be associated with a model architecture from a selection of existing AutoML frameworks, as well as one automatically built model based on the tasks themselves. In parallel, baseline models are generated using an existing AutoML library. Finally, the top performers from the existing AutoML models are combined with the transferred or built models in an ensemble method regime that uses support vector machines (SVM) to yield a final classification output. The framework is tested by performing intelligent fault detection on vibration data from roller element bearings of three separate publicly available datasets.

The framework attained prediction results consistent with other studies in the field, while dramatically reducing the net amount of user interaction time and number of required user decisions. The system also presents the opportunity for self-serve data and model management, helping data scientists spend more time on the creation of new and unique machine learning applications.

5.1 Chapter Introduction

As the need for data-driven decision making and prediction continues to grow in our society, so does the need to harness and make intelligent use of data. Sensor-driven predictive maintenance requires fine tuning models and adjustments to the data strategy used. Lack of labelled data and the rare occurrence of real faults worsen this challenge and lead to a difficult prototyping environment. Those in charge of delivering these insights, data scientists, software engineers, or machine learning engineers, require a great deal of education and experience to build successful models. The field of predictive maintenance involves leveraging sensor data to predict mechanical machine failures before they happen. As previously mentioned, bearings account for approximately 45%-55% of rotating machine failures, and these failures (inner race, outer race, and ball faults) can be predicted by analyzing the vibration signal derived from accelerometers [10]. The challenges in fine tuning and developing customized ML models have fueled the need for automation in the machine learning pipeline, and growth in automated machine learning (AutoML) processes, meta-learning, and transfer learning research. The process of automating machine learning covers a wide range of automation topics, including [86]:

- Data preprocessing;
- Feature extraction;
- Feature engineering;
- Algorithm selection and hyperparameter optimization; and
- Model and data deployment, monitoring, and management.

Work in this area has already begun to achieve highly desirable results. Tuggener et al. demonstrate how AutoML can be used in predictive maintenance for the transport industry, by automatically generating a model that outperforms a team of three data scientists working

for three weeks on the same task [87]. Li et al. combine multiple different datasets with maintenance history to automatically learn rules and select models that result in railway predictive maintenance capabilities with very low false positive rates [88]. Another recent study, and perhaps the closest comparative work to that being done in this thesis, comes from a paper published in 2021 by Lee et al. who developed a transfer learning network for intelligent fault diagnosis based on instance weighting of "practical situations". In their work, a "multi-objective instance weight" was set to determine the proportional influence of domain data compared to the source domain, and used this to guide in the training of the model on the target domain. Their work also sorted situations based on relative similarity between source and test. The IFD in the context of this work used vibration as a data source, as well as time-frequency images (spectrograms) as input to a CNN. This work was tested, however, on an experimental facility setup that involved a robotic arm, servo, jig, and electrode to test the presence of weld faults. Transfer learning in their case increased the fault classification accuracy by 12.3 percentage points over non-transfer learning results. The major difference between this study and the work done in this thesis is that Lee et al. used maximum mean discrepancy (MMD) and Kullback-Leibler Divergence (KLD) working together towards multi-objective instance setting, whereas this work uses KLD to guide training directly.

Many well documented open-source AutoML frameworks exist and are constantly being improved. Auto-sklearn is a Python based framework that primarily relies on Bayesian optimization [89]. Auto-sklearn also handles sparse and missing data, and automatically constructs ensembles. Auto-WEKA also relies on Bayesian optimization, but is built on Java [90]. This automated approach combines multiple search methods and two ensemble methods and was shown to provide competitive performance with standard optimization methods on the CIFAR-10 and MNIST datasets. TPOT is another Python alternative that relies on evolutionary algorithms, essentially mimicking biology in order to breed the best algorithm

for the given task [91]. There are numerous other frameworks, including: Devol (Python), MLBox ML-Plan, SmartML, Autostacker, AlphaD3M, OBOE, PMF, and VDS, but none of these alternatives offer specialization in predictive maintenance [92]. The majority of AutoML libraries do not support neural network architectures, but instead use feature extraction and select from a wide range of traditional machine learning models. Auto-Net is an extension that was added to sklearn for the purpose of expanding its functionality to neural networks. However, dramatic improvements and customizations can be made for applications in predictive maintenance. Most notably, no framework has addressed spectrograms as inputs. Commercial frameworks have also been introduced, including: DataRobot, H20.ai, Google AutoML, SAS Factory Miner, and IBM SPSS Modeler [92]. Nonetheless, the performance benefits of these commercial alternatives are often disputed when compared to open-source libraries. The value of these commercial products tends to revolve around friendly user interfaces and configurable dashboards.

This work focuses on developing and testing an AutoML framework for intelligent fault detection, which takes raw vibration input data, and outputs a final prediction on the presence of a bearing fault. The data preprocessing and feature engineering is automated using an ingestion engine as described in Section 5.2.1.1. The use of transfer learning techniques and automated model building are explained in Section 5.2.2. The combination of these auto-generated models is presented as an ensemble learning output in Section 5.2.3. These three sections represent the sub-modules of the proposed AutoML framework as shown in Figure 30. This system results in the ability to ingest vibration data (and associated fault labels) and automatically build and train models to output a fault prediction. The framework also minimizes the need to consider a large number of parameters shown by previous studies as having a low impact on performance and focus attention, rather, on a few parameters that have a major impact: the size of the data segments, window size, data augmentation

techniques, and fineness of the data distribution samples [77]. This may result in a process that provides reasonable performance and a significantly reduced development time, as the standard approach of persistently tweaking numerous hyperparameters often yields diminishing returns over time.

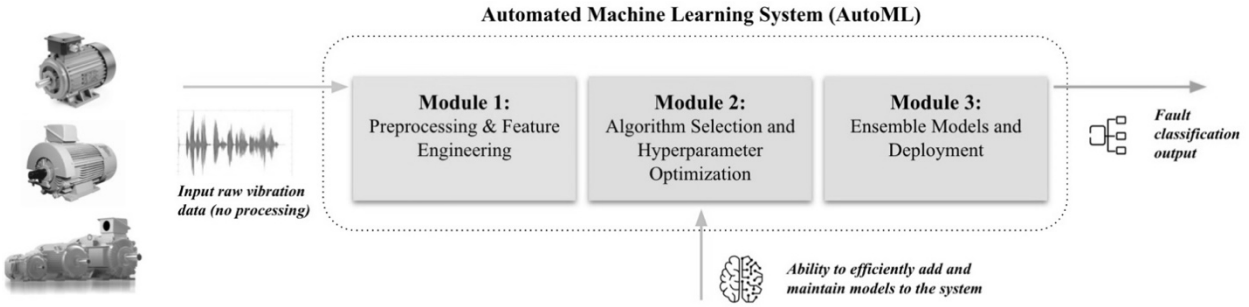


Figure 30: Overview of proposed AutoML framework

5.2 Chapter Methodology

The methodology used for this study uses a sequential progression of operations applied to the system input data. An overview of the framework is provided in Figure 31.

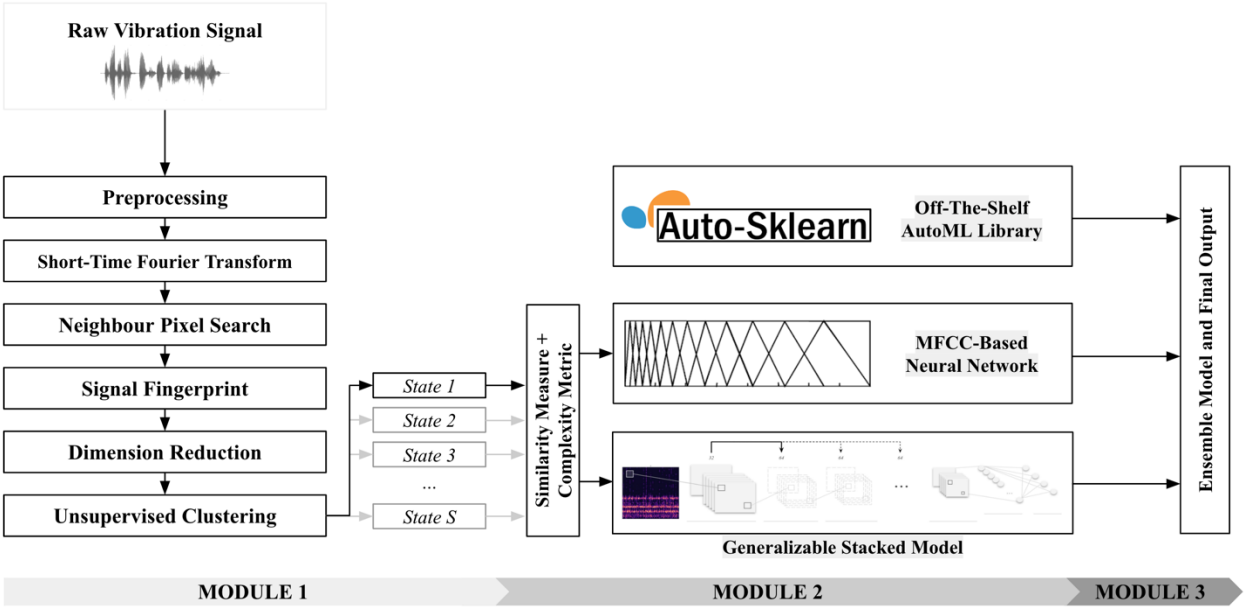


Figure 31: Proposed framework overview.

5.2.1 Module 1: Data Preprocessing

The datasets used in this study involved the CWRU and PbU dataset, as described in Chapter 4, as well as a dataset collected on behalf of the Society for Machinery Failure Prevention Technology (MFPT) that features similar faults and operating speeds to those in the CWRU and PbU datasets. The MFPT dataset also features real world examples, and features publications on both signal processing-based methods, as well as supervised and unsupervised methods [93]. Figure 32 shows the bearings used in the MFPT data collection. The train/test split in this study features the same approach as defined in Section 4.1.1 of Chapter 4.

Figure 32: Inner race and outer race faults on the bearings captured by the MFPT dataset [93].

5.2.1.1 Preprocessing, STFT, and Pre-Augmentation

The first part of the framework seeks to process the data and apply all transformations required for subsequent algorithm selection and optimization processes. This section also serves to standardize the data and provide the user adjustable options, such as window size, overlap, and specific filters that can be applied. The data is resampled at 10kHz, with outlier detection and zero mean then being applied. The short-time Fourier transform (STFT) is then computed on a full-length segment using equation 51. The window size of the STFT is adjustable by the user, but the program defaults to 256 and 50% overlap, resulting in an output of *(129, p)*, where the length of the original sample dictates *p*. There will be a cut-off in which a segment could be abnormally long. In this case, the program will start a new process for it. Otherwise, the program will compute the STFT over the full sample, and then segment the spectrogram into square sizes of (step size + 1) with a total number of segments given by:

$$\left(\frac{p}{step\ size + 1} \times 2\right) + 1 \qquad 54$$

Using floor rounding with the last end discarded, the spectrogram segments are then normalized and labelled before being stored. The visualization of this process output can be seen in Figure 33.

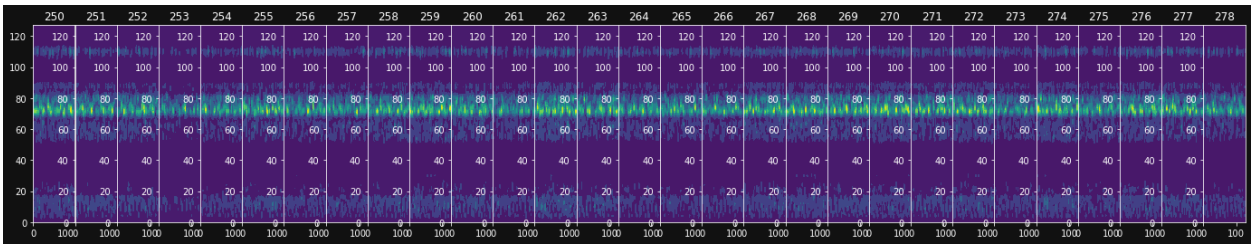


Figure 33: Taking a raw vibration input, computing the STFT and slicing into pieces before sending as input to the program.

5.2.1.2 Fingerprinting and Feature Engineering

With preprocessing, STFT, and pre-augmentation complete, the next step is to separate the data into related operating modes of the machine being observed. These individual groups of data are defined as tasks that the selection algorithm associates most closely with an existing model. The process of doing so involves constructing a fingerprint based on the spectrogram segment from the prior process and is shown in Figure 13 (left). The fingerprint is constructed by extracting time-frequency peaks within the spectrogram, as explained in Section 3.2.1. This strategy was chosen due to its robustness and consistency in dealing with noise and other problematic signals within the spectrogram. The peaks are valuable time-frequency data features that describe the operation of the machine. In traditional machine learning literature, statistical features such as root mean square, skewness, kurtosis, entropy and others, as covered by Pinedo-Sánchez et al [94], are used as inputs for a classifier. Because the goal in this case is to separate machine states, the fingerprints will be added to these features. The vectors together will form a matrix of feature vectors representing the signal, as shown in Figure 34.

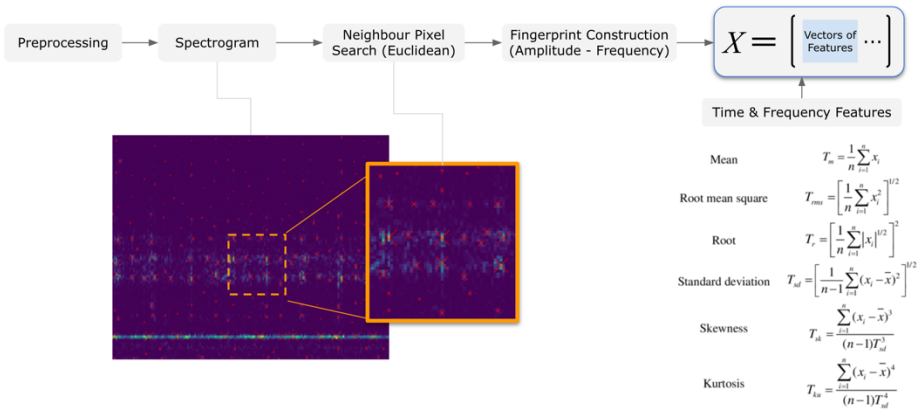


Figure 34: Matrix X, consisting of feature vectors from the vibration signal, similar to Chapter 3.

5.2.1.3 Dimension Reduction and Gaussian Mixture Models

Like Section 3.2.2, the dimension of the feature matrix is reduced using SVD-based PCA. With a matrix of combined frequency peaks, as well as time and frequency features, Gaussian mixture models (GMMs) can then be used as a classification method to group similar task vectors together. They are governed by overlapping clusters with multivariate distributions given by equation 52 defined by a Gaussian distribution, as given in equation 53. Each of these multivariate clusters represent the individual groupings of similar tasks, or in other words, datasets with similar feature representations. The task data will eventually be associated to specific algorithms for training, with this granular understanding of the relative data distribution improving the process of meta-learning. The system outputs a probability score for the data being classified. If scores for incoming data are found to be substantially low when compared to existing task groups, the system creates a new task for that data. Tasks with minimal amounts of data captures at the end of a system run will be grouped together, effectively acting as a "catch-all" to avoid over-complexity.

5.2.2 Module 2: Warm Start, Metalearning and Transfer Learning

Now that data is separated into task groups with similar features organized for each, the process can begin to allocate this data task group to a learning architecture and compute an output. Domain knowledge and previously trained tasks can be leveraged to influence the training of new tasks, which is the focus of meta-learning [95]. In the previous module, tasks were separated from within datasets based on their feature-vector similarity. In this module, the framework will either associate the new data task to an existing model architecture and fine tune it, or will build a new architecture based on the data task features.

First, the data confined to a specific state S is fed to auto-sklearn's library, which uses Bayesian optimization to build a probability-based model that correlates hyperparameter options and resulting performance and uses aid models to guide the search space for future model parameters based on certainty of performance in given space areas. The resulting output is a range of up to 14 traditional machine learning models. Note that this method often fails to attain significant performance on complex tasks, but can often achieve desirable baselines and robust approximations due to the fact that it leverages multiple models. To address this issue, Kullback-Leibler (KL) divergence is used in parallel to measure the data similarity between two datasets and associate similar tasks to either an existing MFCC neural network or a deep learning architecture. The data complexity measurement then guides the addition of more layers. Palanisamy et al. discuss transfer learning on large models such as ResNet and ImageNet, pretrained on the CIFAR dataset, with applications in audio [18], and Hendricks et al. show that hyperparameter selection tends to not result in large changes in performance when testing convolutional networks on spectrograms for predictive maintenance applications [9]. These two studies motivate the approach of selecting

an existing architecture for transfer learning, or simplifying the building of a custom architecture. To assess whether a task is fit for transfer learning on a given model, tasks are mathematically summarized by the feature vector of features described in Figure 34. Tasks given by t_j are assessed for transfer to a new task t_{new}, based on the similarity and distance calculated between the two vectors using KL divergence. Note that the KL divergence calculation must be done on the softmax of t_j and t_{new} vectors, since KL divergence measures similarity of distributions that sum to 1. The task vectors are then compared with existing architectures used on previous data, and the model with the closest match is used with the data. The feature detection layers (convolutional layers) are frozen on all models, and the fully connected layer is retrained with a small and exponentially decaying learning rate. If the task vectors measure a large discrepancy between previous learning tasks, or the user otherwise errs on the side of caution and opts for more redundancy, an iterative stacked learner can be used to build a machine learning model automatically, guided by the similarity of the data within the specific task. This process relies on previous studies that have correlated kernel parameters and CNN architecture under various dataset conditions [46], [77], [95]. These studies have influenced the development of this algorithm, which iteratively stacks small filters on top of each other, covered by a final max pool layer, to build an algorithm with strong precision and generalizability in an automated manner. The algorithm, shown in Figure 35, starts with the input of the task vector and compares itself to similar tasks from previous runs and fetches the number of layers that were successful in that task. Then, the algorithm adds or removes a small filter layer gradually and iteratively, with and without a bottleneck layer. This process is similar for the MFCC network, with the key difference being that in this case it uses the MFCCs (as discussed in Section 2.3) as inputs to an ANN, rather than a CNN.

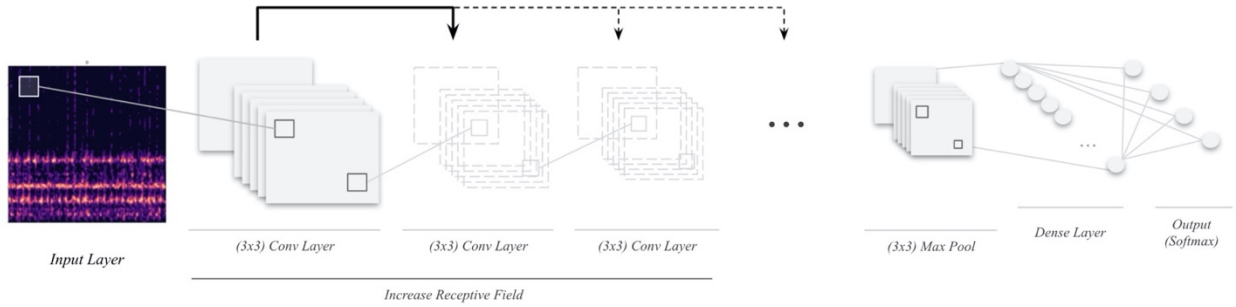


Figure 35: The stacked algorithm process for the CNN.

This process allows for an extremely simple form of search, in which many of the hyperparameter options are fixed, and the width and depth of the network are changed relative to the task vectors and previous runs. The net outcome of this process is an algorithm that can solve for model architecture based both on previous learning tasks, and on domain intuition by knowing which hyperparameters should be changed.

5.2.3 Module 3: Ensemble Methods

The module described in the previous section results in three main outputs: The first is a series of traditional machine learning outputs coming from the auto-sklearn library. The second is the transferred MFCC output, and third is the outputs coming from the transfer learning stacked-based model. Specifically, the outputs from the transfer learning model become the series of training iterations with the kernel values per layer. Together, these outputs form a committee of models. Combinations from within these can improve performance when compared to any of the individual models on their own. Performance improvements are not necessarily limited to accuracy; ensemble methods have been shown to improve variance (usually using bagging), reduce bias (usually using boosting), and result in higher overall accuracy and more stable models (using stacking) [96]. Support vector

machines (SVM) are introduced, which learn to optimize the prediction combinations from each model in the committee to achieve an improved output prediction. This linear classifier improves the final output decision of the framework by taking as input the predictions of the traditional machine learning algorithms, as well as the predictions at each epoch of the transfer learning model, and makes a combined decision. Python's *scikit-Learn* library was used for implementation. The final output of the SVM represents the system prediction. The SVM can also be replaced by simply taking the average of the algorithm prediction, or the top performing prediction, if the user prefers.

5.3 Chapter Results and Discussion

Testing was conducted using a similar arrangement as in Chapter 4: with TensorBoard and Google Colaboratory, K80 GPUs running on off-peak hours. Task specific performances were tracked using a naming convention (taksID-number of samples-…) appended with the final architecture (e.g. 3-layer3x3-bottleneck-16-FClayer-…) facilitating the tracking of results. In Figure 36, task separation (bottom) can be seen along with the BIC criterion for the optimization of number of clusters in the GMM (top). The CWRU dataset was only grouped into two main tasks based on fault severity, as opposed to machine operation. The PbU dataset was divided into more tasks based on operation mode and real versus artificial failures. This lends more weight to the theory that the CWRU dataset presents far less data variance than the other datasets. The MFPT dataset also features tasks with various operation modes. The algorithm favored grouping similar tasks by operating frequency of the machine, which may suggest that this parameter is of higher importance when categorizing machine vibration data; more important than other parameters such as load, torque, or other operating metrics. This has been previously addressed in the literature, with solutions specializing in transient environment and non-stationary operating frequencies [97].

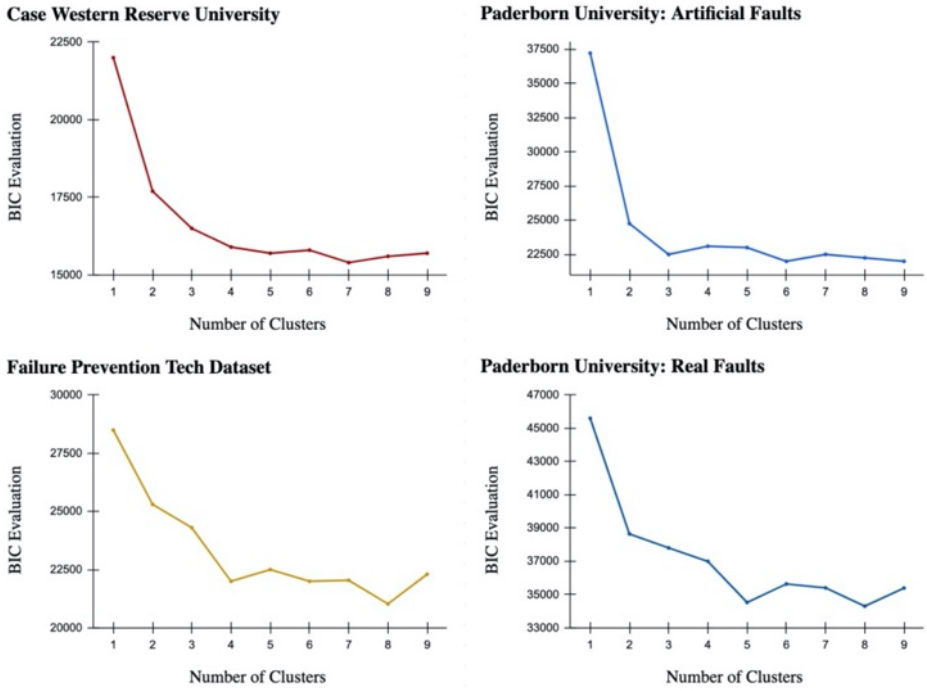


Figure 36: Number of clusters -task- optimization (top), and task association breakdown (bottom) from a single run.

The performances of subsequent associated algorithms when compared to the benchmarks are provided in Table 9. Two separate cases were used with the CWRU dataset. First, the train and test data were separated by horsepower, then by fault severity. The results were compared to a method developed by Zhang et al. using wide first layer kernels (WDCNN) in conjunction with a CNN architecture [75]. The PbU data results were compared to Pandhare et al. who used spectrograms as inputs for their proposed CNN architecture [84]. Similarly, the MFPT data results were compared to the work of Sun et al. using a Hilbert-Huang Transform (HTT) with a CNN [98]. In all cases, the automated method achieves performances within 10% of the target benchmarks.

Table 9: Automated results compared to existing benchmarks.

Dataset	Target Method	Target Benchmark	Automated Result
Case Western Reserve University (Train/test split by horsepower)	WDCNN	90.9%	**84.3%**
Case Western Reserve University (Train/test split by fault severity)	WDCNN	53.1%	**62.5%**
Paderborn University (Artifical)	CNN	95.0%	**87.1%**
Paderborn University (Real)	CNN	61.9%	**71.0%**
MFPT	HHT + CNN	75.9%	**74.5%**

Most of the development time involved in the process was in downloading the datasets and providing them as inputs to the framework. Therefore, future efficiencies can be obtained by ensuring a standardized data format and process. An important observation is that the framework was shown to provide competitive performance on real faults, as well as when data distributions are more challenging. This may be a result of the framework's ability to custom-configure an architecture that is more suitable to a given noisy environment without overfitting due to the constraint on hyperparameter selection. One of the challenges in the system is that the task separation in Module 1 may add unnecessary complexity if there is very large diversity in the data. It could result in frustrating troubleshooting which would defeat the purpose of the program; the user may now have to make decisions on task separation, which could lead to many unwanted train/test cycles. This problem subsides when there is a large amount of data and a large amount of variance, since more of the variance can be captured by the program with fewer clustered groups. Another option (which has not yet been attempted) would be to use a hybrid system where the task separation is a mix of machine learning and manual driven solutions, where dimensional reduction and visualization tools are used to guide the user in decision making.

In this work, the task separation was useful due to differences in the data, in which different model architectures were able to be treated differently. The task separation was

useful in distinguishing between operation modes, which led to customized model building, without the need for human intervention. For example, in the PbU dataset the operation modes were automatically distinguished, and the associated training model was customized accordingly. This, without the need for manual adjustments by a data scientist, allowed for the mitigation of validation error between training and testing datasets where there is potential for memorization and, therefore, a lack of generalization.

A major problem in industry often occurs when lab data is used to train an algorithm with the expectation that performance will remain consistent on real faults in noisy environments. Adaptability is a rare trait when considering solution architecture. The proposed framework has proven its ability to be flexible in the customization of architectures based on new tasks, with the additional bonus of not having to redesign and reconfigure the parameters. The program performs well in separating tasks for the purpose of associating unique architectures to them, leading to stronger performance, and facilitating algorithm reuse. This reduces the friction in rolling out the system on unseen environments and avoids scenarios in which solutions must be redesigned. One of the potential drawbacks of the system is the event in which transfer learning results in worse performance than simple models (formally called negative transfer). This could potentially occur if the unseen data differs greatly from the established theory in both the transfer learning and stacked model making regime. This could also lead to wasted time and resources as models would be trained with suboptimal results. However, since the task association is conducted based on data similarity, with the machine operating frequency being a primary driver for this, there is opportunity for the user to gain deeper insight on the data in an organized and meaningful fashion. There is also an opportunity to implement preprocessing to conduct visual analysis of the data in an effort to guide a human in a judgement call to apply the system to a new application. Data and model management is also conveniently streamlined by the framework. This also leads to the

opportunity for the framework to gradually train larger and more complex models by introducing them to their next nearest task neighbors.

5.4 Chapter Conclusion

In conclusion, this framework dramatically simplifies the model building process by reducing the solution space and minimizing the need for many hyperparameters that would otherwise lead to multiple iterations of parameter tuning or model architecture redesign. Prediction accuracy results that are competitive with current benchmark methods are achieved by the system, with all but one dataset achieving these results within two iterations. By having the system automate redundancy for the user, it provides the user with the opportunity to prioritize their time for deciding on features such as window size, data segment size, filters, and data augmentation techniques, which have more impact on the outcome. Future work would involve leveraging this framework for generalized artificial intelligence. Using task associations in the framework, nearest tasks can be introduced to models to gradually build larger more generalized architectures.

Chapter 6

Thesis Contributions

Chapter 3

- Taking the short time Fourier transform of a vibration signal to obtain the spectrogram, and using it as an input to the unsupervised classification algorithm, provides better information and a standardized data input.
- Changes in operation modes appear on spectrograms, and the time-frequency features can be used to extract differences in these states. This is important because it provides a basis for the grouping of tasks based on similarities.
- By using Gaussian mixture models for clustering, a probability can be assigned to each cluster, which provides a measurement on how much a sample "belongs" to the cluster. Ideally, high probability assignments are desired, but if they aren't, data augmentation and other changes can be made to the program to obtain more appropriate results.

Chapter 4

- Many hyper-parameters do not have a large impact on machine learning model performance. However, certain key architectural decisions, such as kernel sizes, as well as introduction and ordering of pooling layers, have major impacts.
- Stacking smaller square kernels in a 2D CNN without a pooling layer increases the receptive field, which leads to improved feature detection and generalization.

- Introducing a bottleneck layer further forces the network to generalize features, while reducing hyperparameters.
- Competitive results can still be achieved by a generalized model, which results in more robust performance given unseen tasks or noisy environments.

Chapter 5

- KL divergence can be used to measure dataset similarity to pre-existing network datasets and intelligently associate a model architecture.
- This dataset similarity can also be used to guide fine tuning and some hyper-parameter details of 2D CNN models.
- Ensemble models can be used, in combination with transferred networks and off the shelf AutoML frameworks, to yield strong results.
- Combining data similarity measurements, generalized architectures, and ensemble models, a large portion of the deep learning model building can be automated.

In this thesis, a program was built that uses vibration data as input to detect the presence of a bearing fault, with minimal human intervention and design. It can be concluded that many hyperparameters in a convolutional neural network have minimal impact on final classification performance, and for the parameters having a major impact, there are findings that can help guide design decisions. Smaller stacked kernels promote generalization, and bottleneck layers can further force feature abstraction. Furthermore, spectrograms were concluded to be a great input representation not only for their time-frequency domain information, but also their consistent data format and availability for use in 2D CNNs, a powerful deep learning architecture. This work explored using KL divergence as a similarity measure and complexity metric to guide deep learning network architecture decisions.

Nonetheless, there is still much work to be done in formalizing a relationship between KL divergence measurements and the end performance accuracy. Should this program be tested on larger more complex datasets, there is surely areas which would lead to failure or program bugs. There is no mathematical proof that the distance metric and complexity metric set forth in this thesis relates to specific CNN structures; there is only intuition that it does.

A major challenge in scaling this program is the ability to handle larger, more diverse datasets. Within the scope of this thesis, the program only faced 3 equipment configurations. However, 1000s of unique equipment configurations are possible for large fleets of machines operating in real organizations. The challenge continues to be the limited access to datasets in the research community, and when it comes to research and development of deep learning architecture, data is the most important ingredient. Researchers should continue to challenge themselves to construct larger, richer datasets, as well as combine sources when possible.

Chapter 7

Recommendations for Future Work

Currently, much of the published research on IFD for bearings focuses primarily on specific performance benchmarks. However, to enable the proliferation of machine learning in industry, there are other challenges that must be overcome. These include:

- Data standardization
- Model generalization and robustness
- Model deployment

The focus on specific model accuracy benchmarks is relatable, as it provides a single valuable success metric, whereas it can be quite challenging to prove success on other fronts. The work in this thesis focused on the robustness, generalizability, and automation of machine learning for IFD of bearings, which can be quite challenging to measure. Many aspects of machine learning and signal processing had to be used: data augmentation and transformation for the inputs, unsupervised learning for clustering and task segmentation, statistical analysis such as KL divergence for task association, deep learning and transfer learning for model building, and ensemble models for the output. New strategies and algorithms were developed in all these areas.

Future work stemming from this thesis should focus on expanding the preliminary work presented herein in developing a mature program by testing on larger more complex datasets.

Formalizing the mathematical relationship between KL divergence and the end performance accuracy when compared to other potential methods (such as maximum mean discrepancy) would allow for better task segmentation, which in turn leads to higher accuracy performance. Larger datasets also allow more edge cases to be discovered, unique scenarios, or even the possibility of training larger more sophisticated models.

There is a genuine need for the IFD and PdM research community to consolidate and share data and algorithms in a single unified source or framework. Currently, researchers spend precious research time cleaning and prepping data before training new models. If one location could consolidate a data representation and remove that labour, it would accelerate the development of true innovation of unique strategies deployed to turn the data into actionable results, or to facilitate the process of value extraction either by ML or other forms of mathematics. Another possibility for generating more data could involve the use of generative adversarial networks (GANs) to create artificial datasets. However, there is still research needed to prove that GANs can adequately represent real world environments. One concern about artificially created datasets is the potential for unknown strong biases to appear without the knowledge of the authors. Models trained on GAN data and deployed in the real world would need to be validated [99].

Generalized AI is a topic of great interest in predictive maintenance and in many other research areas. Large language models (LLMs) such as ChatGPT have revolutionized the way humans interact with systems and data through text inputs and natural language processing. In the maintenance world, large, generalized models may revolutionize how technicians interact with equipment: instead of text inputs, sensor data could provide a detailed description and understanding of the asset status and health condition. Maintenance of the future will be sensor and information-driven, with minimal guesswork, resulting in the margin of error being significantly reduced.

Another exciting area of focus is augmented reality for maintenance management and work order instructions. Essentially, predictions on faults or equipment failures can be projected to a technician or user via wearables such as a HoloLens or Google glass. These projections can also include guided work instructions to deal with the given faults. Taking this further, there is work being done to completely replicate an asset in the cloud, also known as a digital twin. This digital model captures all the inputs on the equipment in the real world: processes, workflows, sensor data, etc., allowing for a granular understanding and visualization of the equipment's status and health, without needing to be physically present.

References

[1] S. Selcuk, "Predictive maintenance, its implementation and latest trends," *Proc. Inst. Mech. Eng. Part B J. Eng. Manuf.*, vol. 231, no. 9, pp. 1670–1679, Jul. 2017, doi: 10.1177/0954405415601640.

[2] Y. Ran, X. Zhou, P. Lin, Y. Wen, and R. Deng, "A Survey of Predictive Maintenance: Systems, Purposes and Approaches." arXiv, Dec. 12, 2019. doi: 10.48550/arXiv.1912.07383.

[3] "Downtime Costs Auto Industry $22k/Minute - Survey." https://news.thomasnet.com/companystory/downtime-costs-auto-industry-22k-minute-survey-481017 (accessed May 08, 2023).

[4] A. Rai and S. H. Upadhyay, "A review on signal processing techniques utilized in the fault diagnosis of rolling element bearings," *Tribol. Int.*, vol. 96, pp. 289–306, Apr. 2016, doi: 10.1016/j.triboint.2015.12.037.

[5] "Importance of the fourth and fifth intrinsic mode functions for bearing fault diagnosis | IEEE Conference Publication | IEEE Xplore." https://ieeexplore.ieee.org/document/6783140 (accessed Jul. 04, 2023).

[6] C. L. President, "Understanding the Tests that are Recommended for Electric Motor Predictive Maintenance," 2004. Accessed: Jul. 04, 2023. [Online]. Available: https://www.semanticscholar.org/paper/Understanding-the-Tests-that-are-Recommended-for-President/443b732ffd3d9ac2fcf88c834a1f5ba7d1b42d4f

[7] J. T. Renwick and P. E. Babson, "Vibration Analysis—A Proven Technique as a Predictive Maintenance Tool," *IEEE Trans. Ind. Appl.*, vol. IA-21, no. 2, pp. 324–332, Mar. 1985, doi: 10.1109/TIA.1985.349652.

[8] S. A. McInerny and Y. Dai, "Basic vibration signal processing for bearing fault detection," *IEEE Trans. Educ.*, vol. 46, no. 1, pp. 149–156, Feb. 2003, doi: 10.1109/TE.2002.808234.

[9] C. Scheffer and P. Girdhar, *Practical Machinery Vibration Analysis and Predictive Maintenance*. Elsevier, 2004.

[10] T. P. Carvalho, F. A. A. M. N. Soares, R. Vita, R. da P. Francisco, J. P. Basto, and S. G. S. Alcalá, "A systematic literature review of machine learning methods applied to predictive maintenance," *Comput. Ind. Eng.*, vol. 137, p. 106024, Nov. 2019, doi: 10.1016/j.cie.2019.106024.

[11] G. Xu *et al.*, "Data-Driven Fault Diagnostics and Prognostics for Predictive Maintenance: A Brief Overview," in *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, Aug. 2019, pp. 103–108. doi: 10.1109/COASE.2019.8843068.

[12] T. Rieger, S. Regier, I. Stengel, and N. Clarke, "Fast Predictive Maintenance in Industrial Internet of Things (IIoT) with Deep Learning (DL): A Review," *Internet Things*.

[13] G. Cybenkot, "Approximation by superpositions of a sigmoidal function".

[14] R. C. MISHRA and K. PATHAK, *MAINTENANCE ENGINEERING AND MANAGEMENT*. PHI Learning Pvt. Ltd., 2012.

[15] K. Mobley, *Maintenance Engineering Handbook*, 8th Edition. McGraw-Hill Education, 2014. Accessed: May 08, 2023. [Online]. Available: https://www.accessengineeringlibrary.com/content/book/9780071826617

[16] "Rolls-Royce celebrates 50th anniversary of Power-by-the-Hour." https://www.rolls-royce.com/media/press-releases-archive/yr-2012/121030-the-hour.aspx (accessed Jul. 04, 2023).

[17] https://www.bain.com/our-team/pascal-roth, https://www.bain.com/de/our-team/karl-strempel, https://www.bain.com/our-team/oliver-straehle, and https://www.bain.com/our-team/helen-liu, "Machinery as a Service: A Radical Shift Is Underway," *Bain*, May 02, 2022. https://www.bain.com/insights/machinery-as-a-service-global-machinery-and-equipment-report-2022/ (accessed Jul. 04, 2023).

[18] "Equipment-as-a-Service (EaaS): Deloitte whitepaper," *Deloitte Deutschland*. https://www2.deloitte.com/de/de/pages/energy-and-resources/articles/equipment-as-a-service-eaas.html (accessed Jul. 04, 2023).

[19] J. Kim and J.-M. Kim, "Bearing Fault Diagnosis Using Grad-CAM and Acoustic Emission Signals," *Appl. Sci.*, vol. 10, no. 6, Art. no. 6, Jan. 2020, doi: 10.3390/app10062050.

[20] "Application of High-Frequency Resonance Techniques for Bearing Diagnostics in Helicopter Gearboxes." Accessed: May 08, 2023. [Online]. Available: https://apps.dtic.mil/sti/citations/ADA004014

[21] P. D. McFadden, "Examination of a technique for the early detection of failure in gears by signal processing of the time domain average of the meshing vibration," *Mech. Syst. Signal Process.*, vol. 1, no. 2, pp. 173–183, Apr. 1987, doi: 10.1016/0888-3270(87)90069-0.

[22] R. B. Randall, “A New Method of Modeling Gear Faults,” *J. Mech. Des.*, vol. 104, no. 2, pp. 259–267, Apr. 1982, doi: 10.1115/1.3256334.

[23] I. E. Alguindigue, A. Loskiewicz-Buczak, and R. E. Uhrig, “Monitoring and diagnosis of rolling element bearings using artificial neural networks,” *IEEE Trans. Ind. Electron.*, vol. 40, no. 2, pp. 209–217, Apr. 1993, doi: 10.1109/41.222642.

[24] A. Althubaiti, F. Elasha, and J. A. Teixeira, “Fault diagnosis and health management of bearings in rotating equipment based on vibration analysis – a review,” *J. Vibroengineering*, vol. 24, no. 1, Art. no. 1, 2021, doi: 10.21595/jve.2021.22100.

[25] A. Vercoutter, J. Lardies, M. Berthillier, A. Talon, and B. Burgardt, “Improvement of Compressor Blade Vibrations Spectral Analysis From Tip Timing Data: Aliasing Reduction,” presented at the ASME Turbo Expo 2013: Turbine Technical Conference and Exposition, American Society of Mechanical Engineers Digital Collection, Nov. 2013. doi: 10.1115/GT2013-96016.

[26] R. Yan and R. X. Gao, “Energy-Based Feature Extraction for Defect Diagnosis in Rotary Machines,” *IEEE Trans. Instrum. Meas.*, vol. 58, no. 9, pp. 3130–3139, Sep. 2009, doi: 10.1109/TIM.2009.2016886.

[27] N. T. van der Merwe and A. J. Hoffman, “A modified cepstrum analysis applied to vibrational signals,” in *2002 14th International Conference on Digital Signal Processing Proceedings. DSP 2002 (Cat. No.02TH8628)*, Jul. 2002, pp. 873–876 vol.2. doi: 10.1109/ICDSP.2002.1028229.

[28] Z. Kh. Abdul and A. K. Al-Talabani, “Mel Frequency Cepstral Coefficient and its Applications: A Review,” *IEEE Access*, vol. 10, pp. 122136–122158, 2022, doi: 10.1109/ACCESS.2022.3223444.

[29] R. X. Gao, R. Yan, S. Sheng, and L. Zhang, “Sensor Placement and Signal Processing for Bearing Condition Monitoring,” in *Condition Monitoring and Control for Intelligent Manufacturing*, L. Wang and R. X. Gao, Eds., in Springer Series in Advanced Manufacturing. London: Springer, 2006, pp. 167–191. doi: 10.1007/1-84628-269-1_7.

[30] R. F. R. Junior, I. A. dos S. Areias, M. M. Campos, C. E. Teixeira, L. E. B. da Silva, and G. F. Gomes, “Fault detection and diagnosis in electric motors using 1d convolutional neural networks with multi-channel vibration signals,” *Measurement*, vol. 190, p. 110759, Feb. 2022, doi: 10.1016/j.measurement.2022.110759.

[31] J. Brownlee, “4 Types of Classification Tasks in Machine Learning,” *MachineLearningMastery.com*, Apr. 07, 2020.

https://machinelearningmastery.com/types-of-classification-in-machine-learning/ (accessed May 08, 2023).

[32] W. Sun, J. Chen, and J. Li, "Decision tree and PCA-based fault diagnosis of rotating machinery," *Mech. Syst. Signal Process.*, vol. 21, no. 3, pp. 1300–1317, Apr. 2007, doi: 10.1016/j.ymssp.2006.06.010.

[33] "Fault diagnosis of spur bevel gear box using discrete wavelet features and Decision Tree classification - Amrita Vishwa Vidyapeetham," Feb. 28, 2020. https://www.amrita.edu/publication/fault-diagnosis-of-spur-bevel-gear-box-using-discrete-wavelet-features-and-decision-tree-classification/ (accessed Feb. 20, 2023).

[34] "Bearing Data Center | Case School of Engineering | Case Western Reserve University," *Case School of Engineering*, Aug. 05, 2021. https://engineering.case.edu/bearingdatacenter (accessed May 08, 2023).

[35] D. Forsyth, *Applied Machine Learning*. Cham: Springer International Publishing, 2019. doi: 10.1007/978-3-030-18114-7.

[36] S. Rohrmanstorfer, M. Komarov, and F. Mödritscher, "Image Classification for the Automatic Feature Extraction in Human Worn Fashion Data," *Mathematics*, vol. 9, no. 6, Art. no. 6, Jan. 2021, doi: 10.3390/math9060624.

[37] P. L. Fernández-Cabán, F. J. Masters, and B. M. Phillips, "Predicting Roof Pressures on a Low-Rise Structure From Freestream Turbulence Using Artificial Neural Networks," *Front. Built Environ.*, vol. 4, 2018, Accessed: May 09, 2023. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fbuil.2018.00068

[38] M. Fuchs, *Roger Federer vs. Deep Learning: can AI predict Federer's serve ?* 2018.

[39] B. A. Paya, I. I. Esat, and M. N. M. Badi, "ARTIFICIAL NEURAL NETWORK BASED FAULT DIAGNOSTICS OF ROTATING MACHINERY USING WAVELET TRANSFORMS AS A PREPROCESSOR," *Mech. Syst. Signal Process.*, vol. 11, no. 5, pp. 751–765, Sep. 1997, doi: 10.1006/mssp.1997.0090.

[40] P. K. Kankar, S. C. Sharma, and S. P. Harsha, "Vibration-based fault diagnosis of a rotor bearing system using artificial neural network and support vector machine," *Int. J. Model. Identif. Control*, vol. 15, no. 3, pp. 185–198, Jan. 2012, doi: 10.1504/IJMIC.2012.045691.

[41] N. E. Helwig, "Data, Covariance, and Correlation Matrix".

[42] S. L. Brunton and J. N. Kutz, *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*, 2 edition. Cambridge, United Kingdom, New York, NY: Cambridge University Press, 2022.

[43] Y. Hong, M. Kim, H. Lee, J. J. Park, and D. Lee, "Early Fault Diagnosis and Classification of Ball Bearing Using Enhanced Kurtogram and Gaussian Mixture Model," *IEEE Trans. Instrum. Meas.*, vol. 68, no. 12, pp. 4746–4755, Dec. 2019, doi: 10.1109/TIM.2019.2898050.

[44] M.-S. Yang, C.-Y. Lai, and C.-Y. Lin, "A robust EM clustering algorithm for Gaussian mixture models," *Pattern Recognit.*, vol. 45, no. 11, pp. 3950–3961, Nov. 2012, doi: 10.1016/j.patcog.2012.04.031.

[45] "What is a Convolutional Neural Network?," *NVIDIA Data Science Glossary*. https://www.nvidia.com/en-us/glossary/data-science/convolutional-neural-network/ (accessed Jun. 16, 2023).

[46] J. Pons, T. Lidy, and X. Serra, "Experimenting with musically motivated convolutional neural networks," in *2016 14th International Workshop on Content-Based Multimedia Indexing (CBMI)*, Jun. 2016, pp. 1–6. doi: 10.1109/CBMI.2016.7500246.

[47] S. J. Pan and Q. Yang, "A Survey on Transfer Learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010, doi: 10.1109/TKDE.2009.191.

[48] R. Caruana, "Multitask Learning," *Mach. Learn.*, vol. 28, no. 1, pp. 41–75, Jul. 1997, doi: 10.1023/A:1007379606734.

[49] Z. Wang, Z. Dai, B. Póczos, and J. Carbonell, "Characterizing and Avoiding Negative Transfer," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019, pp. 11285–11294. doi: 10.1109/CVPR.2019.01155.

[50] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2014. Accessed: Jun. 16, 2023. [Online]. Available: https://proceedings.neurips.cc/paper/2014/hash/375c71349b295fbe2dcdca9206f20a06-Abstract.html

[51] A. Alekseev and A. Bobe, "GaborNet: Gabor filters with learnable parameters in deep convolutional neural networks." arXiv, Apr. 30, 2019. Accessed: Jun. 16, 2023. [Online]. Available: http://arxiv.org/abs/1904.13204

[52] *Tutorial 74 - What are Gabor filters and how to use them to generate features for machine learning?*, (Sep. 14, 2020). Accessed: Jun. 16, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=yn1NUwaxhZg

[53] "CS 230 - Convolutional Neural Networks Cheatsheet." https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks (accessed Jun. 16, 2023).

[54] N. Adaloglou, "Understanding the receptive field of deep convolutional networks," *AI Summer*, Jul. 02, 2020. https://theaisummer.com/receptive-field/ (accessed Jun. 16, 2023).

[55] M. S. Azari, F. Flammini, S. Santini, and M. Caporuscio, "A Systematic Literature Review on Transfer Learning for Predictive Maintenance in Industry 4.0," *IEEE Access*, vol. 11, pp. 12887–12910, 2023, doi: 10.1109/ACCESS.2023.3239784.

[56] Y. Zhang, C. Bingham, Z. Yang, B. W.-K. Ling, and M. Gallimore, "Machine fault detection by signal denoising—with application to industrial gas turbines," *Measurement*, vol. 58, pp. 230–240, Dec. 2014, doi: 10.1016/j.measurement.2014.08.020.

[57] M. Quiñones-Grueiro, A. Prieto-Moreno, C. Verde, and O. Llanes-Santiago, "Data-driven monitoring of multimode continuous processes: A review," *Chemom. Intell. Lab. Syst.*, vol. 189, pp. 56–71, Jun. 2019, doi: 10.1016/j.chemolab.2019.03.012.

[58] H. Kekre, N. Bhandari, N. Nair, P. Padmanabhan, and S. Bhandari, "A Review of Audio Fingerprinting and Comparison of Algorithms," *Int. J. Comput. Appl.*, vol. 70, pp. 24–30, May 2013, doi: 10.5120/12022-8054.

[59] Y. Zhang, C. Bingham, M. Martínez-García, and D. Cox, "Detection of Emerging Faults on Industrial Gas Turbines Using Extended Gaussian Mixture Models," *Int. J. Rotating Mach.*, vol. 2017, p. e5435794, May 2017, doi: 10.1155/2017/5435794.

[60] "Vibration-based Condition Monitoring: Industrial, Automotive and Aerospace Applications, 2nd Edition | Wiley," *Wiley.com*. https://www.wiley.com/en-sg/Vibration+based+Condition+Monitoring%3A+Industrial%2C+Automotive+and+Aerospace+Applications%2C+2nd+Edition-p-9781119477556 (accessed Jun. 19, 2023).

[61] D. Abboud, S. Baudin, J. Antoni, D. Rémond, M. Eltabach, and O. Sauvage, "The spectral analysis of cyclo-non-stationary signals," *Mech. Syst. Signal Process.*, vol. 75, pp. 280–300, Jun. 2016, doi: 10.1016/j.ymssp.2015.09.034.

[62] F. Scholkmann, J. Boss, and M. Wolf, "An Efficient Algorithm for Automatic Peak Detection in Noisy Periodic and Quasi-Periodic Signals," *Algorithms*, vol. 5, no. 4, Art. no. 4, Dec. 2012, doi: 10.3390/a5040588.

[63] A. M. Colak, Y. Shibata, and F. Kurokawa, “FPGA implementation of the automatic multiscale based peak detection for real-time signal analysis on renewable energy systems,” in *2016 IEEE International Conference on Renewable Energy Research and Applications (ICRERA)*, Nov. 2016, pp. 379–384. doi: 10.1109/ICRERA.2016.7884365.

[64] M. Duarte, “Notes on Scientific Computing for Biomechanics and Motor Control.” Jul. 06, 2023. Accessed: Jul. 09, 2023. [Online]. Available: https://github.com/demotu/BMC

[65] R. Rodomansky, “Fingerprint audio files & identify what’s playing.” Jul. 06, 2023. Accessed: Jul. 09, 2023. [Online]. Available: https://github.com/itspoma/audio-fingerprint-identifying-python

[66] S. W. Choi, J. H. Park, and I.-B. Lee, “Process monitoring using a Gaussian mixture model via principal component analysis and discriminant analysis,” *Comput. Chem. Eng.*, vol. 28, no. 8, pp. 1377–1387, Jul. 2004, doi: 10.1016/j.compchemeng.2003.09.031.

[67] Q. Jiang, B. Huang, and X. Yan, “GMM and optimal principal components-based Bayesian method for multimode fault diagnosis,” *Comput. Chem. Eng.*, vol. 84, pp. 338–349, Jan. 2016, doi: 10.1016/j.compchemeng.2015.09.013.

[68] H. Huang, N. Baddour, and M. Liang, “Short-Time Kurtogram for Bearing Fault Feature Extraction Under Time-Varying Speed Conditions,” presented at the ASME 2018 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, American Society of Mechanical Engineers Digital Collection, Nov. 2018. doi: 10.1115/DETC2018-85165.

[69] G. Orzech, “Intelligent Systems Division,” *NASA*, Aug. 13, 2020. http://www.nasa.gov/intelligent-systems-division (accessed Jun. 19, 2023).

[70] L. Ma, J. Dong, and K. Peng, “Root cause diagnosis of quality-related faults in industrial multimode processes using robust Gaussian mixture model and transfer entropy,” *Neurocomputing*, vol. 285, pp. 60–73, Apr. 2018, doi: 10.1016/j.neucom.2018.01.028.

[71] “How AI Affects the Future Predictive Maintenance: A Primer of Deep Learning,” *springerprofessional.de*. https://www.springerprofessional.de/en/how-ai-affects-the-future-predictive-maintenance-a-primer-of-dee/15457830 (accessed Jul. 02, 2023).

[72] D. Neupane and J. Seok, “Bearing Fault Detection and Diagnosis Using Case Western Reserve University Dataset With Deep Learning Approaches: A Review,” *IEEE Access*, vol. 8, pp. 93155–93178, 2020, doi: 10.1109/ACCESS.2020.2990528.

[73] Y. Chen, G. Peng, C. Xie, W. Zhang, C. Li, and S. Liu, "ACDIN: Bridging the gap between artificial and real bearing damages for bearing fault diagnosis," *Neurocomputing*, vol. 294, pp. 61–71, Jun. 2018, doi: 10.1016/j.neucom.2018.03.014.

[74] "Deep Convolutional Transfer Learning Network: A New Method for Intelligent Fault Diagnosis of Machines With Unlabeled Data | IEEE Journals & Magazine | IEEE Xplore." https://ieeexplore.ieee.org/document/8511076 (accessed Jul. 02, 2023).

[75] W. Zhang, G. Peng, C. Li, Y. Chen, and Z. Zhang, "A New Deep Learning Model for Fault Diagnosis with Good Anti-Noise and Domain Adaptation Ability on Raw Vibration Signals," *Sensors*, vol. 17, no. 2, Art. no. 2, Feb. 2017, doi: 10.3390/s17020425.

[76] "Konstruktions- und Antriebstechnik (KAt) - Data Sets and Download (Universität Paderborn)." https://mb.uni-paderborn.de/en/kat/main-research/datacenter/bearing-datacenter/data-sets-and-download (accessed Jul. 02, 2023).

[77] J. Hendriks, P. Dumond, and D. A. Knox, "Towards better benchmarking using the CWRU bearing fault dataset," *Mech. Syst. Signal Process.*, vol. 169, p. 108732, Apr. 2022, doi: 10.1016/j.ymssp.2021.108732.

[78] S. Zhang, S. Zhang, B. Wang, and T. Habetler, *Machine Learning and Deep Learning Algorithms for Bearing Fault Diagnostics - A Comprehensive Review*. 2019.

[79] J. Chen, D. Zhou, Y. Tang, Z. Yang, Y. Cao, and Q. Gu, "Closing the Generalization Gap of Adaptive Gradient Methods in Training Deep Neural Networks." arXiv, Jun. 23, 2020. doi: 10.48550/arXiv.1806.06763.

[80] "CS231n Convolutional Neural Networks for Visual Recognition." https://cs231n.github.io/convolutional-networks/#layersizepat (accessed Jul. 02, 2023).

[81] E. Tzeng, J. Hoffman, N. Zhang, K. Saenko, and T. Darrell, "Deep Domain Confusion: Maximizing for Domain Invariance." arXiv, Dec. 10, 2014. doi: 10.48550/arXiv.1412.3474.

[82] L. Nanni, G. Maguolo, and M. Paci, "Data augmentation approaches for improving animal audio classification," *Ecol. Inform.*, vol. 57, p. 101084, May 2020, doi: 10.1016/j.ecoinf.2020.101084.

[83] W. Brendel and M. Bethge, "Approximating CNNs with Bag-of-local-Features models works surprisingly well on ImageNet." arXiv, Mar. 20, 2019. doi: 10.48550/arXiv.1904.00760.

[84] V. Pandhare, J. Singh, and J. Lee, "Convolutional Neural Network Based Rolling-Element Bearing Fault Diagnosis for Naturally Occurring and Progressing Defects Using

Time-Frequency Domain Features," in *2019 Prognostics and System Health Management Conference (PHM-Paris)*, May 2019, pp. 320–326. doi: 10.1109/PHM-Paris.2019.00061.

[85] K. Palanisamy, D. Singhania, and A. Yao, "Rethinking CNN Models for Audio Classification." arXiv, Nov. 13, 2020. doi: 10.48550/arXiv.2007.11154.

[86] F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., *Automated Machine Learning: Methods, Systems, Challenges*, 1st ed. 2019 edition. Cham, Switzerland: Springer, 2019.

[87] "Automated Machine Learning in Practice: State of the Art and Recent Results - Google Search." https://www.google.com/search?q=Automated+Machine+Learning+in+Practice%3A+State+of+the+Art+and+Recent+Results&rlz=1C5MACD_enCA1042CA1045&oq=Automated+Machine+Learning+in+Practice%3A+State+of+the+Art+and+Recent+Results&aqs=chrome..69i57.688j0j4&sourceid=chrome&ie=UTF-8 (accessed Jul. 02, 2023).

[88] H. Li *et al.*, "Improving rail network velocity: A machine learning approach to predictive maintenance," *Transp. Res. Part C Emerg. Technol.*, vol. 45, pp. 17–26, Aug. 2014, doi: 10.1016/j.trc.2014.04.013.

[89] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Auto-sklearn: Efficient and Robust Automated Machine Learning," in *Automated Machine Learning: Methods, Systems, Challenges*, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., in The Springer Series on Challenges in Machine Learning. Cham: Springer International Publishing, 2019, pp. 113–134. doi: 10.1007/978-3-030-05318-5_6.

[90] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-WEKA: Automatic Model Selection and Hyperparameter Optimization in WEKA," in *Automated Machine Learning: Methods, Systems, Challenges*, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., in The Springer Series on Challenges in Machine Learning. Cham: Springer International Publishing, 2019, pp. 81–95. doi: 10.1007/978-3-030-05318-5_4.

[91] R. S. Olson and J. H. Moore, "TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning," in *Automated Machine Learning: Methods, Systems, Challenges*, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., in The Springer Series on Challenges in Machine Learning. Cham: Springer International Publishing, 2019, pp. 151–160. doi: 10.1007/978-3-030-05318-5_8.

[92] T. Nagarajah and G. Poravi, "A Review on Automated Machine Learning (AutoML) Systems," in *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, Mar. 2019, pp. 1–6. doi: 10.1109/I2CT45611.2019.9033810.

[93] "Fault Data Sets," *Society For Machinery Failure Prevention Technology*. https://www.mfpt.org/fault-data-sets/ (accessed Jul. 02, 2023).

[94] L. A. Pinedo-Sánchez, D. A. Mercado-Ravell, and C. A. Carballo-Monsivais, "Vibration analysis in bearings for failure prevention using CNN," *J. Braz. Soc. Mech. Sci. Eng.*, vol. 42, no. 12, p. 628, Nov. 2020, doi: 10.1007/s40430-020-02711-w.

[95] M. Huisman, J. N. van Rijn, and A. Plaat, "A Survey of Deep Meta-Learning," *Artif. Intell. Rev.*, vol. 54, no. 6, pp. 4483–4541, Aug. 2021, doi: 10.1007/s10462-021-10004-4.

[96] C. Zhang and Y. Ma, Eds., *Ensemble Machine Learning: Methods and Applications*. New York, NY: Springer New York, 2012. doi: 10.1007/978-1-4419-9326-7.

[97] E. Lughofer and M. Sayed-Mouchaweh, Eds., *Predictive Maintenance in Dynamic Systems: Advanced Methods, Decision Support Tools and Real-World Applications*, 1st ed. 2019 edition. New York, NY: Springer, 2019.

[98] G. Sun, Y. Gao, K. Lin, and Y. Hu, "Fine-Grained Fault Diagnosis Method of Rolling Bearing Combining Multisynchrosqueezing Transform and Sparse Feature Coding Based on Dictionary Learning," *Shock Vib.*, vol. 2019, p. e1531079, Nov. 2019, doi: 10.1155/2019/1531079.

[99] O. Serradilla, E. Zugasti, J. Rodriguez, and U. Zurutuza, "Deep learning models for predictive maintenance: a survey, comparison, challenges and prospects," *Appl. Intell.*, vol. 52, no. 10, pp. 10934–10964, Aug. 2022, doi: 10.1007/s10489-021-03004-y.

Appendix: Python Code

2019 ICSV Fingerprinting, Feature Engineering, PCA, and GMM (Python Code)

```
   #This program is shown with uOttawa data, for NASA data please refer the same program
but with the NASA data preparation code in the thesis appendix

import pandas as pd
import matplotlib
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
from os import walk
import scipy.fftpack

import numpy as np
from scipy.signal import detrend
from scipy.ndimage import uniform_filter1d
from scipy.signal import savgol_filter
from scipy.stats import kurtosis, skew
from scipy.ndimage.filters import maximum_filter
from scipy.ndimage.morphology import (generate_binary_structure,
                                      iterate_structure, binary_erosion)
import hashlib
from operator import itemgetter

plt.rcParams['figure.figsize'] = [24, 12]
%matplotlib inline

from scipy.io import loadmat
from scipy.signal import savgol_filter

from sklearn.decomposition import PCA
from sklearn import preprocessing
from sklearn import mixture
from sklearn.mixture import GaussianMixture

#Unpacking the uOttawa Lab data

for firpath, dirnames, filenames in
walk('/content/drive/MyDrive/CodeFromOldMac/Data/LabData/'):
    fileList_lab = filenames

basedir_lab = '/content/drive/MyDrive/CodeFromOldMac/Data/LabData/'
```

```
baseNameForDataFrame_lab = 'File'

dt_lab = pd.DataFrame()

nFiles_lab = len(fileList_lab)


dt_lab = pd.read_csv(basedir_lab+fileList_lab[0],header=None)
dt_lab -= np.mean(dt_lab)
for i in range(1, nFiles_lab):
    dt_lab[str(filenames[i])] = pd.read_csv(basedir_lab+fileList_lab[i],header=None)
    dt_lab[str(filenames[i])] -= np.mean(dt_lab[str(filenames[i])])
dt_lab.head(10)

###seperate test with smaller, split samples########
# dfs = np.array_split(dt_lab, 3)
# df1 = dfs[0]
# df2 = dfs[1]
# df3 = dfs[2]
# df1 = df1.reset_index(drop=True)
# df2 = df2.reset_index(drop=True)
# df3 = df3.reset_index(drop=True)
# dfs = [df1, df2, df3]
# result = pd.concat(dfs, axis=1)
# full_lab_dt = result.fillna(0)
# full_lab_dt


def scale(val, src, dst):
    """
    Scale the given value from the scale of src to the scale of dst.
    """
    return ((val - src[0]) / (src[1]-src[0])) * (dst[1]-dst[0]) + dst[0]

########################################################################
IDX_FREQ_I = 0
IDX_TIME_J = 1

########################################################################
# Sampling rate, related to the Nyquist conditions, which affects
# the range frequencies we can detect.
DEFAULT_FS = 97656

########################################################################
# Size of the FFT window, affects frequency granularity
```

```
DEFAULT_WINDOW_SIZE = 256

################################################################################
# Ratio by which each sequential window overlaps the last and the
# next window. Higher overlap will allow a higher granularity of offset
# matching, but potentially more fingerprints.
DEFAULT_OVERLAP_RATIO = 0.5

################################################################################
# Degree to which a fingerprint can be paired with its neighbors --
# higher will cause more fingerprints, but potentially better accuracy.
DEFAULT_FAN_VALUE = 20

################################################################################
# Minimum amplitude in spectrogram in order to be considered a peak.
# This can be raised to reduce number of fingerprints, but can negatively
# affect accuracy.
DEFAULT_AMP_MIN = 0

################################################################################
# Number of cells around an amplitude peak in the spectrogram in order
# for Dejavu to consider it a spectral peak. Higher values mean less
# fingerprints and faster matching, but can potentially affect accuracy.
PEAK_NEIGHBORHOOD_SIZE = 5

################################################################################
# Thresholds on how close or far fingerprints can be in time in order
# to be paired as a fingerprint. If your max is too low, higher values of
# DEFAULT_FAN_VALUE may not perform as expected.
MIN_HASH_TIME_DELTA = 0
MAX_HASH_TIME_DELTA = 200

################################################################################
# If True, will sort peaks temporally for fingerprinting;
# not sorting will cut down number of fingerprints, but potentially
# affect performance.
PEAK_SORT = True

################################################################################
# Number of bits to throw away from the front of the SHA1 hash in the
# fingerprint calculation. The more you throw away, the less storage, but
# potentially higher collisions and misclassifications when identifying songs.
FINGERPRINT_REDUCTION = 20
```

```
def fingerprint(channel_samples, Fs=DEFAULT_FS,
                wsize=DEFAULT_WINDOW_SIZE,
                wratio=DEFAULT_OVERLAP_RATIO,
                fan_value=DEFAULT_FAN_VALUE,
                    amp_min=DEFAULT_AMP_MIN):
    """
        FFT the channel, log transform output, find local maxima, then return
        locally sensitive hashes.
        """
    # Fs=DEFAULT_FS
    # wsize=DEFAULT_WINDOW_SIZE
    # wratio=DEFAULT_OVERLAP_RATIO
    # fan_value=DEFAULT_FAN_VALUE
    # amp_min=DEFAULT_AMP_MIN
    x = channel_samples

    # FFT the signal and extract frequency components
    arr2D = mlab.specgram(
        channel_samples,
        NFFT=wsize,
        Fs=Fs,
        window=mlab.window_hanning,
        noverlap=int(wsize * wratio))[0]



    # apply log transform since specgram() returns linear array
    #arr2D = 10 * np.log10(arr2D)
    #arr2D[arr2D == -np.inf] = 0  # replace infs with zeros

    # find local maxima
    #local_maxima = get_2D_peaks(arr2D, plot=False, amp_min=amp_min)

    #return local_maxima
    #return generate_hashes(local_maxima, fan_value=fan_value)


    #def get_2D_peaks(arr2D, plot=False, amp_min=DEFAULT_AMP_MIN):
        #
http://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.morphology.iterate_s
tructure.html#scipy.ndimage.morphology.iterate_structure

    struct = generate_binary_structure(2, 1)
    neighborhood = iterate_structure(struct, PEAK_NEIGHBORHOOD_SIZE)
```

```
    # find local maxima using our fliter shape
    local_max = maximum_filter(arr2D, footprint=neighborhood) == arr2D
    background = (arr2D == 0)
    eroded_background = binary_erosion(background, structure=neighborhood,
                                       border_value=1)

    #local_max=local_max.astype(np.float32)
    #eroded_background=eroded_background.astype(np.float32)
    # Boolean mask of arr2D with True at peaks
    detected_peaks = local_max

    #print(get_2D_peaks(dataWork))
    #print(fingerprint(dataWork))

    # extract peaks
    amps = arr2D[(detected_peaks)]
    ampsRatio = scale(amps, (min(amps), max(amps)), (1, 800))

    j, i = np.where(detected_peaks)

    # filter peaks
    amps = amps.flatten()
    peaks = zip(i, j, amps)
    peaks_filtered = [x for x in peaks if x[2] > amp_min]  # freq, time, amp

    # get indices for frequency and time
    frequency_idx = [x[1] for x in peaks_filtered]
    time_idx = [x[0] for x in peaks_filtered]

    return arr2D, i, j, ampsRatio, time_idx, frequency_idx,



#function to plot fingerprints overlayed over spectrogram
def SpecScatter(dataGiven):
    plotdata = fingerprint(dataGiven)
    fig, ax = plt.subplots()
    ax.imshow(plotdata[0])
    plt.scatter(plotdata[1], plotdata[2], s=plotdata[3], color='red', marker='x',
alpha=0.7)
    ax.set_xlabel('Time')
    ax.set_ylabel('Frequency')
    ax.set_title("Spectrogram")
    plt.gca().invert_yaxis()
```

```
    plt.show()


#prep for PCA
scaled_data = preprocessing.scale(normalizedMatrixOfFeatureVectors)
pca = PCA()
pca.fit(scaled_data)
pca_data = pca.transform(scaled_data)

# Store results of PCA in a data frame
result=pd.DataFrame(pca.transform(scaled_data), columns=['PCA%i' % i for i in
range(len(pca_data[0]))])

# Plot initialisation
fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')
ax.scatter(result['PCA0'], result['PCA1'], result['PCA2'], cmap="Set2_r")

# make simple, bare axis lines through space:
# xAxisLine = ((min(result['PCA0']), max(result['PCA0'])), (0, 0), (0,0))
# yAxisLine = ((0, 0), (min(result['PCA1']), max(result['PCA1'])), (0,0))
# zAxisLine = ((0, 0), (0,0), (min(result['PCA2']), max(result['PCA2'])))

# label the axes
ax.set_xlabel("PC1")
ax.set_ylabel("PC2")
ax.set_zlabel("PC3")
#plt.show()

#prep GMM
model = mixture.GaussianMixture(n_components=4, covariance_type='full').fit(result)
labels = model.predict(result)
# plt.scatter(result['PCA0'], result['PCA1'], result['PCA2'], c=labels, s=40,
cmap='viridis')

fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')
ax.set_facecolor('white')
ax.scatter(result['PCA0'], result['PCA1'], result['PCA2'], c=labels)

ax.set_xlabel("PC1")
ax.set_ylabel("PC2")
ax.set_zlabel("PC3")

n_components = np.arange(1, 10)
```

```
models = [GaussianMixture(n, covariance_type='full', random_state=0).fit(result)
          for n in n_components]

plt.plot(n_components, [m.bic(result) for m in models], color='red', marker =
'x',markersize=10, linewidth=2, alpha = 0.9)
plt.legend(loc='best')
plt.xlabel('Number of Clusters')
plt.ylabel('BIC Evaluation')
```

Generalization Focused CNN Architecture Example, ASME 2021 (Python Code)

```
### This example does not include data prepratation. Please see Thesis appendix for data preparation with Case Western and Paderborn Datasets. Also Data exploration python code in appendix of thesis

import json
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow.keras as keras
import tensorflow as tf
import time
from tensorflow.keras.callbacks import TensorBoard
from tensorboard.plugins.hparams import api as hp
import random
from mlxtend.plotting import plot_confusion_matrix
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
import librosa
import matplotlib.pyplot as plt


CW_TRAIN = '/content/drive/MyDrive/MASc/Datasets/MULTICLASS/MULTI_CW_Fault_Severity_data_Train_CNN_7_14.json'
CW_TEST = '/content/drive/MyDrive/MASc/Datasets/MULTICLASS/MULTI_CW_Fault_Severity_data_Test_CNN_21.json'

PB_TRAIN = "/content/drive/MyDrive/MASc/Datasets/MULTICLASS/PB_Artificial_MULTICLASS_SUB.json"
PB_TEST = '/content/drive/MyDrive/MASc/Datasets/MULTICLASS/PB_Real_MULTICLASS_SUB.json'

def load_data(dataset_path):
    with open(dataset_path, "r") as fp:
        data = json.load(fp)

    #Convert lists into numpy arrays
    inputs = np.array(data['Spectrogram'])
    targets = np.array(data['Label'])

    inputs= inputs[..., np.newaxis]

    return inputs, targets
```

```
    ES  =  keras.callbacks.EarlyStopping(monitor='val_accuracy',patience=4,  verbose=0,
mode='max', restore_best_weights=False)

    def data_augmentation(input_array):
      resized_array = []
      for each in input_array:
        noise = np.random.normal(0, .1, each.shape)
        each_noised = each + noise

        if random.random() < 0.5:
          each_noised = np.flip(each_noised,0)
        if random.random() < 0.5:
          each_noised = np.flip(each_noised,1)

        resized_array.append(each_noised)

      each_noised_np = np.asarray(resized_array)

      return each_noised_np
    def step_decay(epoch):
        decay_rate = 1.5
        initial_learning_rate = 0.0001
        learning_rate = initial_learning_rate * (1/(1+(decay_rate*epoch)))
        return learning_rate


    def step_decay_2(epoch):
        decay_rate = 1.25
        initial_learning_rate = 0.00015
        learning_rate = initial_learning_rate * (1/(1+(decay_rate*epoch)))
        return learning_rate
    inputs_train, targets_train = load_data(CW_TRAIN)
    inputs_test, targets_test= load_data(CW_TEST)

    print(inputs_train.shape)
    print(targets_train.shape)
    print('\n')
    print(inputs_test.shape)
    print(targets_test.shape)
    inputs_PB_train, targets_PB_train = load_data(PB_TRAIN)
    inputs_PB_test, targets_PB_test= load_data(PB_TEST)

    print(inputs_PB_train.shape)
```

```
    print(targets_PB_train.shape)
    print('\n')
    print(inputs_PB_test.shape)
    print(targets_PB_test.shape)
    # inputs_train_3 = data_augmentation(inputs_train)
    # inputs_test_3 = data_augmentation(inputs_test)

    inputs_PB_train_3 = data_augmentation(inputs_PB_train)
    inputs_PB_test_3 = data_augmentation(inputs_PB_test)


    # inputs_train_3 = np.repeat(inputs_train_3, 3, -1)
    # inputs_test_3 = np.repeat(inputs_test_3, 3, -1)

    # print(inputs_train_3.shape)
    # print(inputs_test_3.shape)

    print(inputs_PB_train_3.shape)
    print(inputs_PB_test_3.shape)



    keras.backend.clear_session()
    # Build the CNN net
    input_shape = (inputs_train.shape[1], inputs_train.shape[2], inputs_train.shape[3])
#From before: 4D array -> (num_samples, 128, 128, 1)

    t = time.localtime()

    model_square_aug2 = keras.Sequential()

    # 1st conv layer
    model_square_aug2.add(keras.layers.Conv2D(8,     (3,3),     strides     =     (1,1),
activation='relu', input_shape=input_shape, kernel_regularizer = 'l2')) #Conv2D(kernal,
grid size, activation, )
    #model_square_aug2.add(keras.layers.Conv2D(32,     (3,3),     strides     =     (2,2),
activation='relu', input_shape=input_shape, kernel_regularizer = 'l2')) #Conv2D(kernal,
grid size, activation, )
    model_square_aug2.add(keras.layers.Conv2D(16,     (3,3),     strides     =     (1,1),
activation='relu', input_shape=input_shape, kernel_regularizer = 'l2')) #Conv2D(kernal,
grid size, activation, )
    model_square_aug2.add(keras.layers.MaxPool2D((2,2),  strides  =  (2,2),  padding  =
'same'))
    model_square_aug2.add(keras.layers.BatchNormalization()) # Process that normalizes
the activations in a layer (speeds up training, models are more reliable)
```

```
   model_square_aug2.add(keras.layers.Conv2D(4,     (1,1),     strides     =     (1,1),
activation='relu', input_shape=input_shape, kernel_regularizer = 'l2')) #Conv2D(kernal,
grid size, activation, )
   model_square_aug2.add(keras.layers.Conv2D(4,     (3,3),     strides     =     (1,1),
activation='relu', input_shape=input_shape, kernel_regularizer = 'l2')) #Conv2D(kernal,
grid size, activation, )
   model_square_aug2.add(keras.layers.Conv2D(4,     (1,1),     strides     =     (1,1),
activation='relu', input_shape=input_shape, kernel_regularizer = 'l2')) #Conv2D(kernal,
grid size, activation, )
   #model_square_aug2.add(keras.layers.MaxPool2D((2,2),  strides  =  (2,2),  padding  =
'same'))


   # Flatten the output and feed it into dense layer
   model_square_aug2.add(keras.layers.Flatten())
   model_square_aug2.add(keras.layers.Dense(16, activation = 'relu', kernel_regularizer
= 'l2')) # 64 neurons
   model_square_aug2.add(keras.layers.Dropout(0.4))

   # Output layer
   model_square_aug2.add(keras.layers.Dense(4, activation = 'softmax'))

   # Compile the network
   optimizer = keras.optimizers.Adam(learning_rate = 0.0001)
   model_square_aug2.compile(optimizer        =        optimizer,        loss        =
'sparse_categorical_crossentropy', metrics = ['accuracy'])

   model_square_aug2.summary()
   lr_rate = keras.callbacks.LearningRateScheduler(step_decay)

   model_square_aug2.fit(inputs_PB_train_3, targets_PB_train, batch_size = 32, epochs =
30, callbacks = [lr_rate])

   model_square_weights_aug2 = []
   for i in range(0,9):
     model_square_weights_aug2.append(model_square_aug2.layers[i].get_weights())


   ####### FOR TRANSFER LEARNING ########
   # Simply need to save weights with
   for i in range(0,'number of layers in model'):
   model_name_new_model.layers[i].set_weights(model_name_old_model.layers[i].get_weigh
ts())
   # Before calling .fit on new model. Can optionally play with freezing layers as well
```

KL Divergence Measurement Example (Python Code)

```
   uOttawaLab_KL_Divergence = []
   NASA_KL_Divergence = []
   SoftmaxOfFeatureVectors_uOttawa = []
   SoftmaxOfFeatureVectors_NASA = []

   #KL Divergence only works if datasets being compared individually sum to 1
   for j in range(0, len(NASAnormalizedMatrixOfFeatureVectors.columns), 1):

SoftmaxOfFeatureVectors_NASA.append((softmax(NASAnormalizedMatrixOfFeatureVectors.iloc
[:,j])))

   for j in range(0, len(normalizedMatrixOfFeatureVectors.columns), 1):

SoftmaxOfFeatureVectors_uOttawa.append((softmax(normalizedMatrixOfFeatureVectors.iloc[
:,j])))

   #datasets being compared must also be the same length
   for i in range(0, len(SoftmaxOfFeatureVectors_uOttawa[0]), 1)
     if len(SoftmaxOfFeatureVectors_NASA[i]) < len(SoftmaxOfFeatureVectors_uOttawa[i]):
       SoftmaxOfFeatureVectors_uOttawa[i]    =    SoftmaxOfFeatureVectors_uOttawa[i][:
len(SoftmaxOfFeatureVectors_NASA[i])]
     elif                    len(SoftmaxOfFeatureVectors_NASA[i])                    >
len(SoftmaxOfFeatureVectors_uOttawa[i]):
       SoftmaxOfFeatureVectors_NASA[i]       =       SoftmaxOfFeatureVectors_NASA[i][:
len(SoftmaxOfFeatureVectors_uOttawa[i])]

   #Can now compute the KL divergence of either a single sample, or the average KLD over
a dataset
   KL_Divergence_Sample           =           kl_div(SoftmaxOfFeatureVectors_NASA[0],
SoftmaxOfFeatureVectors_uOttawa[0])
   KL_Divergence_Sample = 1 - sum(KL_Divergence_Sample)

   KL_Divergence_Dataset = []
   for i in range(0, len(SoftmaxOfFeatureVectors_NASA),1):
     KL_single               =               kl_div(SoftmaxOfFeatureVectors_NASA[i],
SoftmaxOfFeatureVectors_uOttawa[i])
     KL_single = 1 - sum(KL_single)
     KL_Divergence_Dataset.append(KL_single)
```

Iterative Model Building Example (after KL Divergence) ROSE 2021, (Python Code)

```
   # Iteratively Build the CNN net
   input_shape = (inputs_train.shape[1], inputs_train.shape[2], inputs_train.shape[3])
#From before: 4D array -> (num_samples, 128, 128, 1)

   #These values will be chosen based on the KL Divergence of incoming data
   layer_depths = [8,16,32]
   kernel_sizes = [(3,3),(5,5),(7,7)]
   conv_layers = [1,2,3]
   count = 0
   t = time.localtime()
   for layer_depth in layer_depths:
     for kernel_size in kernel_sizes:
       for conv_layer in conv_layers:
         for i in range(1):

           model = keras.Sequential()

           # 1st conv layer
           model.add(keras.layers.Conv2D(layer_depth, kernel_size, strides = (2,2),
activation='relu', input_shape=input_shape, kernel_regularizer = 'l2')) #Conv2D(kernal,
grid size, activation, )
           model.add(keras.layers.MaxPool2D((3,3), strides = (2,2), padding = 'same'))
           model.add(keras.layers.BatchNormalization()) # Process that normalizes the
activations in a layer (speeds up training, models are more reliable)

           for i in range(conv_layer - 1):
             model.add(keras.layers.Conv2D(layer_depth, kernel_size, strides = (2,2),
activation='relu', input_shape=input_shape, kernel_regularizer = 'l2')) #Conv2D(kernal,
grid size, activation, )
             model.add(keras.layers.MaxPool2D((3,3), strides = (2,2), padding =
'same'))
             model.add(keras.layers.BatchNormalization()) # Process that normalizes the
activations in a layer (speeds up training, models are more reliable)

           # Flatten the output and feed it into dense layer
           model.add(keras.layers.Flatten())
           model.add(keras.layers.Dense(8, activation = 'relu')) # 64 neurons
           model.add(keras.layers.Dropout(0.3))

           # Output layer
```

```
            model.add(keras.layers.Dense(4, activation = 'softmax'))

            # Compile the network
            optimizer = keras.optimizers.Adam(learning_rate = 0.0001)
            model.compile(optimizer         =         optimizer,         loss         =
'sparse_categorical_crossentropy', metrics = ['accuracy'])

            hp_number = round(model.count_params(), -4)

            NAME    =    '{}-depth-{}-kernel-{}-conv-{}-hp-{}-run-{}'.format(layer_depth,
kernel_size, conv_layer, hp_number, time.strftime('%d-%H-%M-%S', t), str(i))
            tensorboard                                                                =
TensorBoard(log_dir='/content/drive/MyDrive/MASc/TensorBoardLogs/CWPB/SQUARE/{}'.forma
t(NAME), update_freq='epoch')
            count += 1
            print(NAME)

            model.fit(inputs_train,     targets_train,     validation_data=(inputs_test,
targets_test), batch_size = 32, epochs = 40, callbacks = [tensorboard])

            keras.backend.clear_session()
```

MFCC Data Preparation and MLP Example, ROSE 2021 (Python Code)

```
import json
import numpy as np
import librosa
from sklearn.model_selection import train_test_split
import tensorflow.keras as keras
import matplotlib.pyplot as plt
import math
import scipy.io as sc
from os import walk
plt.rcParams['figure.figsize'] = [24, 12]


#####GLOBAL_VARIABLES############

FAULT_DATASET_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/MechDatasets/CaseWestern_Organized/All/Drive'
BASELINE_DATASET_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/MechDatasets/CaseWestern_Organized/All/Baseline
/Normal_1_1772.mat'
JSON_PATH = "CW_data.json"

TOTAL_NUMBER_OF_SAMPLES = 120000
SAMPLING_FREQ = 12000
NUMBER_OF_SEGMENTS = 100
SAMPLES_PER_SEGMENT = int(TOTAL_NUMBER_OF_SAMPLES/NUMBER_OF_SEGMENTS)

n_mfcc = 13
n_fft = 1024
hop_length = 256

num_mfcc_vectors_per_segment = math.ceil(SAMPLES_PER_SEGMENT/hop_length)

data = {'Mapping': [],
       'Label':[],
       'MFCC':[]
       }

################################
def retrieve_signal_from_mat_file(file_path):

    # load the files into matfile
    matfile = sc.loadmat(file_path)

    #Grab the keys
```

```
    keys = list(matfile.keys())

    #Initialize a keyword search
    keyword = ['','']

    #Find the Drive End data array in each sample
    for each in keys:
        if "DE" in each:
            keyword = str(each) #if it finds it, set the keyword as the file title

    signal = matfile[keyword] #This extracts the specific sample
    signal = list(signal[:,0]) #turns it into a list

    return signal


def mfcc_on_sample(signal, hot_label, map_name):

    #We want to grab the mfccs on each segment, so, looping through the segments
    for d in range(NUMBER_OF_SEGMENTS):
        start = SAMPLES_PER_SEGMENT*d
        finish = start + SAMPLES_PER_SEGMENT

        mfcc = librosa.feature.mfcc(np.array(signal[start:finish]), SAMPLING_FREQ,
n_mfcc=n_mfcc, n_fft=n_fft, hop_length=hop_length)
        mfcc = mfcc.T

        # Checking and then appending the mfccs of each segment to the data
dictionary, along with the label
        if len(mfcc) == num_mfcc_vectors_per_segment:
            data['MFCC'].append(mfcc.tolist())
            data['Label'].append(hot_label)
            print("{}, segment:{}".format(map_name, d+1))
    print('\n\n')

def save_mfcc():

    #First, setting up the baseline data

    #Get the signal
    baseline_signal = retrieve_signal_from_mat_file(BASELINE_DATASET_PATH)

    #Calculate mfccs and append with labels
    mfcc_on_sample(baseline_signal, hot_label=0, map_name='Baseline')
    data['Mapping'].append('Baseline')
```

```
    #Grab list of filenames for the other readings (the readings with the seeded
faults)
    for dirpath, dirnames, filenames in walk(FAULT_DATASET_PATH):
        fileList = filenames

    #Loop through fileList
    for each in fileList:
        matches = ['_1', '021']
        if all(x in each for x in matches): #When a file is 1HP and 0.021 fault


#         if "_1" in each:
#             if "021" in each:
            #Set up the map name and associated label for the right fault
            if "IR" in each:
                map_name = 'Inner'
                hot_label = 1
            elif "B" in each:
                map_name = 'Ball'
                hot_label = 2
            elif "@6" in each:
                map_name = 'Center'
                hot_label = 3
            elif "@3" in each:
                map_name = 'Orthogonal'
                hot_label = 4
            elif "@12" in each:
                continue

            else:
                print('PROBLEM1')

            print(each)

            #Grab the signal from the file
            signal_path = FAULT_DATASET_PATH + '/' + each
            signal = retrieve_signal_from_mat_file(signal_path)

            #Append the mapping name
            data['Mapping'].append(map_name)
            print("\nProcessing: {}".format(map_name))

            #Calculate mfccs and append with labels
            mfcc_on_sample(signal, hot_label=hot_label, map_name=map_name)
```

```
    with open(JSON_PATH, "w") as fp:
        json.dump(data, fp, indent=4)

        #Another File Walk for Baseline
save_mfcc()


##########

DATASET_PATH = "/Users/justinlarocque-
villiers/Desktop/CodeFolder/Sandbox/Signal_Processing/CW_data.json"

# load data
# load data

def load_data(dataset_path):
    with open(dataset_path, "r") as fp:
        data = json.load(fp)

    #Convert lists into numpy arrays
    inputs = np.array(data['MFCC'])
    targets = np.array(data['Label'])

    return inputs, targets

def plot_history(history):

    fig, axs = plt.subplots(2)

    #Create accuracy subplot
    axs[0].plot(history.history['accuracy'], label='Train Accuracy')
    axs[0].plot(history.history['val_accuracy'], label='Test Accuracy')
    axs[0].set_ylabel('Accuracy')
    axs[0].legend(loc='lower right')
    axs[0].set_title('Accuracy Eval')

    #Create error subplot
    axs[1].plot(history.history['loss'], label='Train Error')
    axs[1].plot(history.history['val_loss'], label='Test Error')
    axs[1].set_ylabel('Error')
    axs[1].set_xlabel('Epoch')
    axs[1].legend(loc='upper right')
    axs[1].set_title('Error Eval')
```

```
    plt.show()


inputs, targets = load_data(DATASET_PATH)

# split data into train and test sets
inputs_train, inputs_test, targets_train, targets_test = train_test_split(inputs,
targets, test_size = 0.3)

# Build the network architectures
model = keras.Sequential([
    #input layer
    keras.layers.Flatten(input_shape = (inputs.shape[1], inputs.shape[2])),

    #1st hidden layer
    keras.layers.Dense(512, activation='relu',
kernel_regularizer=keras.regularizers.l2(0.001)),
    keras.layers.Dropout(0.3),

    #2nd hidden layer
    keras.layers.Dense(256, activation='relu',
kernel_regularizer=keras.regularizers.l2(0.001)),
    keras.layers.Dropout(0.3),

    #3rd hidden layer
    keras.layers.Dense(64, activation='relu',
kernel_regularizer=keras.regularizers.l2(0.001)),
    keras.layers.Dropout(0.3),

    #Output layer
    keras.layers.Dense(5, activation = 'softmax')
])

# compile network
optimizer = keras.optimizers.Adam(learning_rate=0.00008)
model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics = ['accuracy'])

model.summary()


# train network
history = model.fit(inputs_train, targets_train,
           validation_data=(inputs_test, targets_test),
```

```
          epochs = 40,
          batch_size=32)

# plot accuracy and eror over the epochs
plt.style.use('dark_background')
    plot_history(history)
```

AutoSKLearn Example Using Paderborn Data, ROSE 2021 (Python Code)

```
!pip install Cython numpy
!pip install scikit-learn --upgrade
!pip install auto-sklearn --upgrade
!pip install dask distributed --upgrade
!pip install pipelineprofiler


import autosklearn.classification
import sklearn.model_selection
import sklearn.datasets
import sklearn.metrics

import json
import numpy as np
import time
import random


PB_TRAIN                                                                    =
"/content/drive/MyDrive/MASc/Datasets/MULTICLASS/PB_Artificial_MULTICLASS_SUB.json"
PB_TEST                                                                     =
'/content/drive/MyDrive/MASc/Datasets/MULTICLASS/PB_Real_MULTICLASS_SUB.json'

def load_data(dataset_path):
    with open(dataset_path, "r") as fp:
        data = json.load(fp)

    #Convert lists into numpy arrays
    inputs = np.array(data['Spectrogram'])
    targets = np.array(data['Label'])

    inputs= inputs[..., np.newaxis]

    return inputs, targets
inputs_PB_train, targets_PB_train = load_data(PB_TRAIN)
inputs_PB_test, targets_PB_test= load_data(PB_TEST)

print(inputs_PB_train.shape)
print(targets_PB_train.shape)
print('\n')
print(inputs_PB_test.shape)
print(targets_PB_test.shape)
from skimage.transform import rescale, resize, downscale_local_mean

resize_factor = 4
```

```
    def resize_it(input_array):
      resized_array = []
      for each in input_array:
        each_resized = resize(each, (each.shape[0] // resize_factor, each.shape[1] //
resize_factor))
        resized_array.append(each_resized)
      resized_array_np = np.asarray(resized_array)
      return resized_array_np


    def Flatten(input_not_flat):
      flat_array = []
      for each in input_not_flat:
        flat_array.append(each.flatten())
      flat_array_np = np.asarray(flat_array)
      return flat_array_np

    inputs_PB_test_r = resize_it(inputs_PB_test[:,:,:,0])
    inputs_PB_test_r.shape
    inputs_PB_train_r = resize_it(inputs_PB_train[:,:,:,0])
    inputs_PB_train_r.shape
    inputs_PB_train_f= Flatten(inputs_PB_train_r)

    #targets_train_f= Flatten(targets_train)
    inputs_PB_test_f= Flatten(inputs_PB_test_r)
    #targets_test_f= Flatten(targets_test)

    print(inputs_PB_train_f.shape)
    print(inputs_PB_test_f.shape)
    print(targets_PB_train.shape)
    print(targets_PB_test.shape)

    automl                                                                    =
autosklearn.classification.AutoSklearnClassifier(time_left_for_this_task=400,
per_run_time_limit=60) #Auto-sklearn searches pipelines for 1 minute
    automl.fit(inputs_PB_train_f, targets_PB_train)
    y_hat = automl.predict(inputs_PB_test_f)

    print("Accuracy score", sklearn.metrics.accuracy_score(targets_PB_test, y_hat))
    print(automl.sprint_statistics())

    import PipelineProfiler
    data = PipelineProfiler.import_autosklearn(automl)
    PipelineProfiler.plot_pipeline_matrix(data)
```

Paderborn Data Preparation Example for Both Real and Artificial Faults (Python Code)

```
import json
import numpy as np
import librosa
from sklearn.model_selection import train_test_split
import tensorflow.keras as keras
import matplotlib.pyplot as plt
import math
import scipy.io as sc
from os import walk
import os
from scipy import signal

from skimage.transform import rescale, resize, downscale_local_mean

from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

plt.rcParams['figure.figsize'] = [24, 12]

#####GLOBAL_VARIABLES############

TRAIN_SAVE_PATH = "/Users/justinlarocque-
villiers/Desktop/CodeFolder/PB_Artificial_MULTICLASS_FULL.json"
TEST_SAVE_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/PB_Real_MULTICLASS_FULL.json'

TRAIN_BASELINE_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_TWOCLASS/PB_READY_TRAIN/Basel
ine'
TRAIN_INNER_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_TWOCLASS/PB_READY_TRAIN/Inner
'
TRAIN_OUTER_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_TWOCLASS/PB_READY_TRAIN/Outer
'

TEST_BASELINE_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_TWOCLASS/PB_READY_TEST/Baseli
ne'
TEST_INNER_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_TWOCLASS/PB_READY_TEST/Inner'
```

```
TEST_OUTER_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_TWOCLASS/PB_READY_TEST/Outer'

DATA_SF = 64000
REQUIRED_SF = 48000
WINDOWSIZE = 128
STEPSIZE = 64

data_test = {'Mapping': ['Baseline', 'Faulty'],
       'Label':[],
       'Spectrogram':[]
       }
################################
def retrieve_signal_from_mat_file(file_path, file_name):

    matfile = sc.loadmat(file_path)
    base_key = os.path.splitext(file_name)[0]
    data_file=matfile[str(base_key)]['Y'][0][0][0][6][2]

    data = data_file[0]

    new_number_of_samples = (len(data)*REQUIRED_SF)//DATA_SF

    resampled_data = signal.resample(data, new_number_of_samples)
    resampled_data -= np.mean(resampled_data)

    return resampled_data


def spectrogram_on_sample(signal, hot_label, map_name):

    S = librosa.stft(np.array(signal), n_fft=WINDOWSIZE, hop_length=STEPSIZE)
    S_scale = np.abs(S)
    num_rows = S_scale.shape[0]
    num_cols = S_scale.shape[1]
    num_steps = num_cols//num_rows
    print('Processing new sample\n')
    print('Spectrogram sample shape: {}\n'.format(S_scale.shape))
    print('Broken into {} pieces\n'.format(num_steps))

    resize_factor = S_scale.shape[0]/128

    for i in range(0, num_steps):
        S_square = S_scale[:,int(i*num_rows):int((i+1)*num_rows)]
        normalized_square = scaler.fit_transform(S_square)
```

```
        S_normalized_resized = resize(S_square, (S_square.shape[0] // resize_factor,
S_square.shape[1] // resize_factor),
                        anti_aliasing=True)

        if 128 not in (S_normalized_resized.shape[0], S_normalized_resized.shape[1]):
            print('################# PROBLEM PROBLEM PROBLEM PROBLEM
##################')


        data_test['Spectrogram'].append(S_normalized_resized.tolist())
        data_test['Label'].append(hot_label)
        print("{}, segment:{}".format(map_name, i+1))
        print("Shape: {}".format(S_normalized_resized.shape))


    print('\n\n')

def save_spectrograms_train():

    ##### OUTER RACE ########

    for dirpath, dirnames, filenames in walk(TRAIN_OUTER_PATH):
        fileList = filenames


    fileList[:] = [x for x in fileList if "Store" not in x]

    for each in fileList:
        signal_path = TRAIN_OUTER_PATH + '/' + each

        signal = retrieve_signal_from_mat_file(signal_path, each)
        spectrogram_on_sample(signal, hot_label=3, map_name='Center')


    ##### INNER RACE ########

    for dirpath, dirnames, filenames in walk(TRAIN_INNER_PATH):
        fileList = filenames

    fileList[:] = [x for x in fileList if "Store" not in x]

    for each in fileList:
        signal_path = TRAIN_INNER_PATH + '/' + each

        signal = retrieve_signal_from_mat_file(signal_path, each)
```

```
        spectrogram_on_sample(signal, hot_label=2, map_name='Inner')

    ##### HEALTHY ########

    for dirpath, dirnames, filenames in walk(TRAIN_BASELINE_PATH):
        fileList = filenames

    fileList[:] = [x for x in fileList if "Store" not in x]

    for each in fileList[:10]:
        signal_path = TRAIN_BASELINE_PATH + '/' + each

        signal = retrieve_signal_from_mat_file(signal_path, each)
        spectrogram_on_sample(signal, hot_label=0, map_name='Baseline')


    with open(TRAIN_SAVE_PATH, "w") as fp:
        json.dump(data_test, fp, indent=4)


def save_spectrograms_test():

    ##### OUTER RACE ########

    for dirpath, dirnames, filenames in walk(TEST_OUTER_PATH):
        fileList = filenames


    fileList[:] = [x for x in fileList if "Store" not in x]

    for each in fileList:
        signal_path = TEST_OUTER_PATH + '/' + each

        signal = retrieve_signal_from_mat_file(signal_path, each)
        spectrogram_on_sample(signal, hot_label=3, map_name='Center')


    ##### INNER RACE ########

    for dirpath, dirnames, filenames in walk(TEST_INNER_PATH):
        fileList = filenames

    fileList[:] = [x for x in fileList if "Store" not in x]

    for each in fileList:
```

```
        signal_path = TEST_INNER_PATH + '/' + each

        signal = retrieve_signal_from_mat_file(signal_path, each)
        spectrogram_on_sample(signal, hot_label=2, map_name='Inner')

   ##### HEALTHY ########

    for dirpath, dirnames, filenames in walk(TEST_BASELINE_PATH):
        fileList = filenames

    fileList[:] = [x for x in fileList if "Store" not in x]

    for each in fileList[:10]:
        signal_path = TEST_BASELINE_PATH + '/' + each

        signal = retrieve_signal_from_mat_file(signal_path, each)
        spectrogram_on_sample(signal, hot_label=0, map_name='Baseline')


    with open(TEST_SAVE_PATH, "w") as fp:
        json.dump(data_test, fp, indent=4)


save_spectrograms_train()

save_spectrograms_test()



   #### Paderborn artificial faults ####

   import json
import numpy as np

import librosa
from sklearn.model_selection import train_test_split
import tensorflow.keras as keras
import matplotlib.pyplot as plt
import math
import scipy.io as sc
from os import walk
import os
from scipy import signal

from skimage.transform import rescale, resize, downscale_local_mean
```

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

plt.rcParams['figure.figsize'] = [24, 12]

#####GLOBAL_VARIABLES############

TRAIN_SAVE_PATH = "/Users/justinlarocque-
villiers/Desktop/CodeFolder/PB_Artificial_TWOCLASS_FULL.json"
TEST_SAVE_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/PB_Real_TWOCLASS_FULL.json'

TRAIN_BASELINE_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_FULL/PB_READY_TRAIN/Baseline'
TRAIN_INNER_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_FULL/PB_READY_TRAIN/Inner'
TRAIN_OUTER_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_FULL/PB_READY_TRAIN/Outer'

TEST_BASELINE_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_FULL/PB_READY_TEST/Baseline'
TEST_INNER_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_FULL/PB_READY_TEST/Inner'
TEST_OUTER_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/PB_FULL/PB_READY_TEST/Outer'

DATA_SF = 64000
REQUIRED_SF = 48000
WINDOWSIZE = 128
STEPSIZE = 64

data_test = {'Mapping': ['Baseline', 'Faulty'],
       'Label':[],
       'Spectrogram':[]
       }
################################
def retrieve_signal_from_mat_file(file_path, file_name):

    matfile = sc.loadmat(file_path)
    base_key = os.path.splitext(file_name)[0]
    data_file=matfile[str(base_key)]['Y'][0][0][0][6][2]

    data = data_file[0]

    new_number_of_samples = (len(data)*REQUIRED_SF)//DATA_SF
```

```
    resampled_data = signal.resample(data, new_number_of_samples)
    resampled_data -= np.mean(resampled_data)

    return resampled_data


def spectrogram_on_sample(signal, hot_label, map_name):

    S = librosa.stft(np.array(signal), n_fft=WINDOWSIZE, hop_length=STEPSIZE)
    S_scale = np.abs(S)
    num_rows = S_scale.shape[0]
    num_cols = S_scale.shape[1]
    num_steps = num_cols//num_rows
    print('Processing new sample\n')
    print('Spectrogram sample shape: {}\n'.format(S_scale.shape))
    print('Broken into {} pieces\n'.format(num_steps))

    resize_factor = S_scale.shape[0]/128

    for i in range(0, num_steps):
        S_square = S_scale[:,int(i*num_rows):int((i+1)*num_rows)]
        normalized_square = scaler.fit_transform(S_square)
        S_normalized_resized = resize(S_square, (S_square.shape[0] // resize_factor,
S_square.shape[1] // resize_factor),
                      anti_aliasing=True)

        if 128 not in (S_normalized_resized.shape[0], S_normalized_resized.shape[1]):
            print('################ PROBLEM PROBLEM PROBLEM PROBLEM
##################')


        data_test['Spectrogram'].append(S_normalized_resized.tolist())
        data_test['Label'].append(hot_label)
        print("{}, segment:{}".format(map_name, i+1))
        print("Shape: {}".format(S_normalized_resized.shape))


    print('\n\n')

def save_spectrograms_train():

    ##### OUTER RACE ########

    for dirpath, dirnames, filenames in walk(TRAIN_OUTER_PATH):
```

```
        fileList = filenames


    fileList[:] = [x for x in fileList if "Store" not in x]

    for each in fileList:
        signal_path = TRAIN_OUTER_PATH + '/' + each

        signal = retrieve_signal_from_mat_file(signal_path, each)
        spectrogram_on_sample(signal, hot_label=1, map_name='Faulty')


    ##### INNER RACE ########

    for dirpath, dirnames, filenames in walk(TRAIN_INNER_PATH):
        fileList = filenames

    fileList[:] = [x for x in fileList if "Store" not in x]

    for each in fileList:
        signal_path = TRAIN_INNER_PATH + '/' + each

        signal = retrieve_signal_from_mat_file(signal_path, each)
        spectrogram_on_sample(signal, hot_label=1, map_name='Faulty')

   ##### HEALTHY ########

    for dirpath, dirnames, filenames in walk(TRAIN_BASELINE_PATH):
        fileList = filenames

    fileList[:] = [x for x in fileList if "Store" not in x]

    for each in fileList[:10]:
        signal_path = TRAIN_BASELINE_PATH + '/' + each

        signal = retrieve_signal_from_mat_file(signal_path, each)
        spectrogram_on_sample(signal, hot_label=0, map_name='Baseline')


    with open(TRAIN_SAVE_PATH, "w") as fp:
        json.dump(data_test, fp, indent=4)

def save_spectrograms_test():

    ##### OUTER RACE ########
```

```
 for dirpath, dirnames, filenames in walk(TEST_OUTER_PATH):
     fileList = filenames


 fileList[:] = [x for x in fileList if "Store" not in x]

 for each in fileList:
     signal_path = TEST_OUTER_PATH + '/' + each

     signal = retrieve_signal_from_mat_file(signal_path, each)
     spectrogram_on_sample(signal, hot_label=1, map_name='Faulty')


 ##### INNER RACE ########

 for dirpath, dirnames, filenames in walk(TEST_INNER_PATH):
     fileList = filenames

 fileList[:] = [x for x in fileList if "Store" not in x]

 for each in fileList:
     signal_path = TEST_INNER_PATH + '/' + each

     signal = retrieve_signal_from_mat_file(signal_path, each)
     spectrogram_on_sample(signal, hot_label=1, map_name='Faulty')

##### HEALTHY ########

 for dirpath, dirnames, filenames in walk(TEST_BASELINE_PATH):
     fileList = filenames

 fileList[:] = [x for x in fileList if "Store" not in x]

 for each in fileList[:10]:
     signal_path = TEST_BASELINE_PATH + '/' + each

     signal = retrieve_signal_from_mat_file(signal_path, each)
     spectrogram_on_sample(signal, hot_label=0, map_name='Baseline')


 with open(TEST_SAVE_PATH, "w") as fp:
     json.dump(data_test, fp, indent=4)
```

```
save_spectrograms_train()

save_spectrograms_test()
```

Case Western Data Preparation Examples (Python Code)

```
import json
import numpy as np
import librosa
from sklearn.model_selection import train_test_split
import tensorflow.keras as keras
import matplotlib.pyplot as plt
import math
import scipy.io as sc
from os import walk
from scipy import signal

from skimage.transform import rescale, resize, downscale_local_mean

from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

plt.rcParams['figure.figsize'] = [24, 12]

#####GLOBAL_VARIABLES############

FAULT_DATASET_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/CWRU_48DriveEndFault/MULTICLASS/
HP123_Combined'
BASELINE_DATASET_PATH = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/Cross_Datasets/CWRU_48DriveEndFault/MULTICLASS/
Baseline'

JSON_PATH_TRAIN = "CW_SPECTROGRAMS_.json"
JSON_PATH_TEST = "DOWN_MULTI_CW_Fault_Severity_data_Test_CNN_21.json"

DATA_SF = 48000
REQUIRED_SF = 48000
WINDOWSIZE = 128
STEPSIZE = 64


data_train = {'Mapping': [],
       'Label':[],
       'Spectrogram':[]
       }
data_test = {'Mapping': [],
       'Label':[],
       'Spectrogram':[]
```

```
        }
################################

def retrieve_signal_from_mat_file(file_path):

    # load the files into matfile
    matfile = sc.loadmat(file_path)

    #Grab the keys
    keys = list(matfile.keys())
    #print(keys)
    #Initialize a keyword search and data list
    keyword = ['','']
    reading = []

    #Find the Drive End data array in each sample
    for each in keys:
        if "DE" in each:
            keyword[0] = str(each) #if it finds it, set the keyword as the file title
        if "FE" in each:
            keyword[1] = str(each)

    DE_signal = matfile[keyword[0]] #This extracts the specific sample
    DE_signal = list(DE_signal[:,0]) #turns it into a list
    #DE_signal -= np.mean(DE_signal)
    reading.append(DE_signal)

    new_number_of_samples = (len(reading)*REQUIRED_SF)//DATA_SF

    resampled_data = signal.resample(reading, new_number_of_samples)
    resampled_data -= np.mean(resampled_data)

    return resampled_data


def spectrogram_on_sample(signal, hot_label, map_name, set_name):

    S = librosa.stft(np.array(signal), n_fft=WINDOWSIZE, hop_length=STEPSIZE)
    S_scale = np.abs(S)
    num_rows = S_scale.shape[0]
    num_cols = S_scale.shape[1]
    num_steps = num_cols//num_rows
    print('Processing new sample\n')
    print('Spectrogram sample shape: {}\n'.format(S_scale.shape))
    print('Broken into {} pieces\n'.format(num_steps))
```

```
    resize_factor = S_scale.shape[0]/128

    for i in range(0, num_steps):
        #if i > 13: continue
        S_square = S_scale[:,int(i*num_rows):int((i+1)*num_rows)]
        normalized_square = scaler.fit_transform(S_square)
        S_normalized_resized = resize(S_square, (S_square.shape[0] // resize_factor,
S_square.shape[1] // resize_factor),
                      anti_aliasing=True)

        if 128 not in (S_normalized_resized.shape[0], S_normalized_resized.shape[1]):
            print('################ PROBLEM PROBLEM PROBLEM PROBLEM
##################')

        if set_name == 0:
            data_train['Spectrogram'].append(S_normalized_resized.tolist())
            data_train['Label'].append(hot_label)
            print("{}, segment:{}".format(map_name, i+1))
            print("Shape: {}".format(S_normalized_resized.shape))

        elif set_name == 1:
            data_test['Spectrogram'].append(S_normalized_resized.tolist())
            data_test['Label'].append(hot_label)
            print("{}, segment:{}".format(map_name, i+1))
            print("Shape: {}".format(S_normalized_resized.shape))

        else: print('################ PROBLEM PROBLEM PROBLEM PROBLEM
##################')
    print('\n\n')

def save_spectrograms_train():

    #First, setting up the baseline data
    for dirpath, dirnames, filenames in walk(BASELINE_DATASET_PATH):
        fileList = filenames
    for each in fileList:
        if '_0' in each:
            continue
        else:
            #Get the signal
            signal_path = BASELINE_DATASET_PATH + '/' + each

            baseline_signal = retrieve_signal_from_mat_file(signal_path)
            baseline_signal = baseline_signal[0]
```

```
            baseline_signal = baseline_signal[:len(baseline_signal)//2]
            #Calculate spectrograms and append with labels
            spectrogram_on_sample(baseline_signal, hot_label=0, map_name='Baseline',
set_name = 0)

            if 'Baseline' not in data_train['Mapping']:
                data_train['Mapping'].append('Baseline')



    #Grab list of filenames for the other readings (the readings with the seeded
faults)
    for dirpath, dirnames, filenames in walk(FAULT_DATASET_PATH):
        fileList = filenames

    ball_count = 0
    inner_count = 0
    center_count = 0

    #Loop through fileList
    for each in fileList:
        matches_Position = ['007', '014']
        if any(x in each for x in matches_Position):
            print(each)

            #Set up the map name and associated label for the right fault
            if "Ball" in each:
                map_name = 'Ball'
                hot_label = 1
                ball_count += 1
            elif "Inner" in each:
                map_name = 'Inner'
                hot_label = 2
                inner_count += 1
            elif "Outer" in each:
                map_name = 'Center'
                hot_label = 3
                center_count += 1
            else:
                print(each)
                continue

            #Grab the signal from the file
            signal_path = FAULT_DATASET_PATH + '/' + each
            signal = retrieve_signal_from_mat_file(signal_path)
```

```
            #Append the mapping name
            if map_name not in data_train['Mapping']:
                data_train['Mapping'].append(map_name)
            print("\nProcessing: {}".format(map_name))

            #Calculate mfccs and append with labels
            spectrogram_on_sample(signal[0], hot_label=hot_label, map_name=map_name,
set_name = 0)

    print('Ball Count: {}\n'.format(ball_count))
    print('Inner Count: {}\n'.format(inner_count))
    print('Center Count: {}\n'.format(center_count))
    print('\n')
    print('Total Labels: {}'.format(len(data_train['Label'])))
    print('Total Spectrograms: {}'.format(len(data_train['Spectrogram'])))

    with open(JSON_PATH_TRAIN, "w") as fp:
        json.dump(data_train, fp, indent=4)


def save_spectrograms_test():

    #First, setting up the baseline data
    for dirpath, dirnames, filenames in walk(BASELINE_DATASET_PATH):
        fileList = filenames

    for each in fileList:
        if '_0' in each:
            continue
        else:
            #Get the signal
            signal_path = BASELINE_DATASET_PATH + '/' + each

            baseline_signal = retrieve_signal_from_mat_file(signal_path)
            baseline_signal = baseline_signal[0]
            baseline_signal = baseline_signal[len(baseline_signal)//2:]

            #Calculate spectrograms and append with labels
            spectrogram_on_sample(baseline_signal, hot_label=0, map_name='Baseline',
set_name = 1)

            if 'Baseline' not in data_test['Mapping']:
                data_test['Mapping'].append('Baseline')
```

```
    #Grab list of filenames for the other readings (the readings with the seeded
faults)
    for dirpath, dirnames, filenames in walk(FAULT_DATASET_PATH):
        fileList = filenames

    ball_count = 0
    inner_count = 0
    center_count = 0

    #Loop through fileList
    for each in fileList:
        matches_Position = ['021']
        if any(x in each for x in matches_Position):
            print(each)

            #Set up the map name and associated label for the right fault
            if "Ball" in each:
                map_name = 'Ball'
                hot_label = 1
                ball_count += 1
            elif "Inner" in each:
                map_name = 'Inner'
                hot_label = 2
                inner_count += 1
            elif "Outer" in each:
                map_name = 'Center'
                hot_label = 3
                center_count += 1
            else:
                print(each)
                continue

            #Grab the signal from the file
            signal_path = FAULT_DATASET_PATH + '/' + each
            signal = retrieve_signal_from_mat_file(signal_path)

            #Append the mapping name
            if map_name not in data_train['Mapping']:
                data_train['Mapping'].append(map_name)
            print("\nProcessing: {}".format(map_name))

            #Calculate mfccs and append with labels
            spectrogram_on_sample(signal[0], hot_label=hot_label, map_name=map_name,
set_name = 1)
```

```
    print('Ball Count: {}\n'.format(ball_count))
    print('Inner Count: {}\n'.format(inner_count))
    print('Center Count: {}\n'.format(center_count))
    print('\n')
    print('Total Labels: {}'.format(len(data_test['Label'])))
    print('Total Spectrograms: {}'.format(len(data_test['Spectrogram'])))

    with open(JSON_PATH_TEST, "w") as fp:
        json.dump(data_test, fp, indent=4)


save_spectrograms_train()

save_spectrograms_test()
```

University of Ottawa and NASA Dataset Preparation Example (Python Code)

```
### uOttawa Dataset ###

## Need appropriate imports (not listed here for sake of redundancy)

#Unpacking the uOttawa Lab data

for firpath, dirnames, filenames in
walk('/content/drive/MyDrive/CodeFromOldMac/Data/LabData/'):
    fileList_lab = filenames

basedir_lab = '/content/drive/MyDrive/CodeFromOldMac/Data/LabData/'
baseNameForDataFrame_lab = 'File'

dt_lab = pd.DataFrame()

nFiles_lab = len(fileList_lab)


dt_lab = pd.read_csv(basedir_lab+fileList_lab[0],header=None)
dt_lab -= np.mean(dt_lab)
for i in range(1, nFiles_lab):
    dt_lab[str(filenames[i])] = pd.read_csv(basedir_lab+fileList_lab[i],header=None)
    dt_lab[str(filenames[i])] -= np.mean(dt_lab[str(filenames[i])])
dt_lab.head(10)

dfs = np.array_split(dt_lab, 3)
df1 = dfs[0]
df2 = dfs[1]
df3 = dfs[2]
df1 = df1.reset_index(drop=True)
df2 = df2.reset_index(drop=True)
df3 = df3.reset_index(drop=True)
dfs = [df1, df2, df3]
result = pd.concat(dfs, axis=1)
full_lab_dt = result.fillna(0)
full_lab_dt


matrixOfFeatureVectors = []

for i in range(0, len(full_lab_dt.columns),1):

  sampleFeatureVector = []
  rootMeanSquare = np.sqrt(np.mean(full_lab_dt.iloc[:,i]**2))
```

```
  kurtosisValue = kurtosis(full_lab_dt.iloc[:,i])
  skewnessValue = skew(full_lab_dt.iloc[:,i])
  peakToPeak = np.ptp(full_lab_dt.iloc[:,i])
  crestFactor = abs(peakToPeak)/rootMeanSquare
  shapeFactor =
rootMeanSquare/(sum(abs(full_lab_dt.iloc[:,i]))/len(full_lab_dt.iloc[:,i]))

  sampleFeatureVector.extend((rootMeanSquare, kurtosisValue, skewnessValue,
peakToPeak, crestFactor, shapeFactor))

  finderprintData = fingerprint(full_lab_dt.iloc[:,i])
  dt_fingerprint = pd.DataFrame()
  dt_fingerprint["i"] = finderprintData[1]
  dt_fingerprint["j"] = finderprintData[2]

  fringerprintVectorMagnitude = []

  for j in range(0,10,1):
      fringerprintVectorMagnitude.append(np.linalg.norm([finderprintData[1][j],
finderprintData[2][j]]))

  sampleFullVector = sampleFeatureVector + fringerprintVectorMagnitude
  matrixOfFeatureVectors.append(sampleFullVector)


matrixOfFeatureVectors_dt = pd.DataFrame(matrixOfFeatureVectors)

normalizedMatrixOfFeatureVectors =(matrixOfFeatureVectors_dt-
matrixOfFeatureVectors_dt.min())/(matrixOfFeatureVectors_dt.max()-
matrixOfFeatureVectors_dt.min())


### NASA Dataset ###

#Unpacking the NASA dataset

   for             firpath,             dirnames,             filenames             in
walk('/content/drive/MyDrive/CodeFromOldMac/Data/1st_test'):
       fileList = filenames

   #there are 2156 files
   basedir = '/content/drive/MyDrive/CodeFromOldMac/Data/1st_test/'
   baseNameForDataFrame = 'File'

   NASA_dt = pd.DataFrame()
```

```
    nFiles = len(fileList)
    count = 0

    for i in range(0,len(fileList),5):
        df_time_dump = pd.read_csv(basedir+fileList[i], sep='\t', header=None)
        NASA_dt['File'      +      str(i)      ]=df_time_dump[df_time_dump.columns[0]]-
np.mean(df_time_dump[df_time_dump.columns[0]]) #only take the 1st column (x axis)
    NASA_dt.head(10)


    NASAmatrixOfFeatureVectors = []

    for i in range(0, len(NASA_dt.columns),1):

      NASAsampleFeatureVector = []
      rootMeanSquare = np.sqrt(np.mean(NASA_dt.iloc[:,i]**2))
      kurtosisValue = kurtosis(NASA_dt.iloc[:,i])
      skewnessValue = skew(NASA_dt.iloc[:,i])
      peakToPeak = np.ptp(NASA_dt.iloc[:,i])
      crestFactor = abs(peakToPeak)/rootMeanSquare
      shapeFactor = rootMeanSquare/(sum(abs(NASA_dt.iloc[:,i]))/len(NASA_dt.iloc[:,i]))

      NASAsampleFeatureVector.extend((rootMeanSquare,   kurtosisValue,   skewnessValue,
peakToPeak, crestFactor, shapeFactor))

      finderprintData = fingerprint(NASA_dt.iloc[:,i])
      dt_fingerprint = pd.DataFrame()
      dt_fingerprint["i"] = finderprintData[1]
      dt_fingerprint["j"] = finderprintData[2]

      NASAfringerprintVectorMagnitude = []

      for j in range(0,20,1):
          NASAfringerprintVectorMagnitude.append(np.linalg.norm([finderprintData[1][j],
finderprintData[2][j]]))

      NASAsampleFullVector = NASAsampleFeatureVector + NASAfringerprintVectorMagnitude
      NASAmatrixOfFeatureVectors.append(NASAsampleFullVector)


    NASAmatrixOfFeatureVectors_dt = pd.DataFrame(NASAmatrixOfFeatureVectors)
    NASAnormalizedMatrixOfFeatureVectors               =(NASAmatrixOfFeatureVectors_dt-
NASAmatrixOfFeatureVectors_dt.min())/(NASAmatrixOfFeatureVectors_dt.max()-
NASAmatrixOfFeatureVectors_dt.min())
```

MFTP Data Exploration Example (Python Code)

```
import numpy as np
import pandas as pd
import math
import scipy.io as sc
from scipy.signal import hilbert

import json
from os import walk

import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = [24,12]
plt.style.use('dark_background')


FAULT_DATASET_PATH = ''
BASELINE_DATASET_PATH = ''

matfile = sc.loadmat('/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/MechDatasets/MFPT Fault Data Sets/3 - Seven
More Outer Race Fault Conditions/OuterRaceFault_vload_2.mat')

keys = list(matfile.keys())
file = matfile['bearing']
file = file[0]
file = file[0]
file[2]
plt.plot(file[2])


jacobfile = sc.loadmat('/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/MechDatasets/CaseWestern_Organized/unified_data
.mat')
keys = list(jacobfile.keys())
jacobsfile = jacobfile['unified_data']
jacobsfile = jacobsfile[0]
jacobsfile = jacobsfile[0]

unified_HP1 = jacobsfile[0]
unified_HP2 = jacobsfile[1]
unified_HP3 = jacobsfile[2]

words = unified_HP1[:,1]
```

```
#print(unified_HP1[:,1])

words[0]

def retrieve_signal_from_mat_file(file_path):

    # load the files into matfile
    matfile = sc.loadmat(file_path)

    #Grab the keys
    keys = list(matfile.keys())

    #Initialize a keyword search
    keyword = ['','']

    #Find the Drive End data array in each sample
    for each in keys:
        if "DE" in each:
            keyword = str(each) #if it finds it, set the keyword as the file title

    signal = matfile[keyword] #This extracts the specific sample
    signal = list(signal[:,0]) #turns it into a list

    return retrieved_data

#Grab list of filenames for the other readings (the readings with the seeded faults)
for dirpath, dirnames, filenames in walk(FAULT_DATASET_PATH):
    fileList = filenames

#Loop through fileList
for each in fileList:


    #Grab the signal from the file
    signal_path = FAULT_DATASET_PATH + '/' + each
    retrieved_data = retrieve_signal_from_mat_file(signal_path)
```

Misc Python Code for Exploring the Datasets, Feature Engineering, Peak Finding, and Fingerprinting

```
from __future__ import division, print_function
from os import walk
import pandas as pd
import matplotlib
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

%matplotlib inline
plt.rcParams['figure.figsize'] = [32, 16]

from scipy.signal import detrend
from scipy.ndimage import uniform_filter1d
from scipy.signal import savgol_filter

from scipy.ndimage.filters import maximum_filter
from scipy.ndimage.morphology import (generate_binary_structure,
                                      iterate_structure, binary_erosion)
import hashlib
from operator import itemgetter

from scipy.io import loadmat


for firpath, dirnames, filenames in walk('/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/MechDatasets/IMS/1st_test'):
    fileList = filenames
#there are 2156 files


basedir = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/MechDatasets/IMS/1st_test/'
baseName = 'File'
nFiles = len(filenames)
step = 10
df2 = pd.DataFrame()
df3 = pd.DataFrame()

def shortDataGrab(nFiles):
    count = 0
```

```
    for i in range(0,nFiles):
        df1 = pd.read_csv(basedir+fileList[i], sep='\t', header=None)
        df2['File ' + str(i) ]=df1[df1.columns[0]]
        # First we take the absolutes of the fast fourier transform
        amplitude = np.absolute(np.fft.fft(df2['File ' + str(i)]))

        # Now we ignore the 2nd half of the transform as being complex conjugates of
the 1st half
        amplitude = amplitude[0:(int(len(amplitude)/2))]
        amplitude[0]=0
        #amplitude = savgol_filter(amplitude,21,1)
        df3['File ' + str(i) ]=amplitude
        count += 1
        #for i in range(fileRange, len(filenames))
    return df3

print(nFiles)

# Number of samplepoints
N = 20480
# sample spacing
T = 1.0 / 20000
x = np.linspace(0.0, N*T, N)
xf = np.linspace(0.0, 1.0/(2.0*T), N/2)

 # First we take the absolutes of the fast fourier transform
pre_yf = np.absolute(np.fft.fft(df2['File 1']))

# Now we ignore the 2nd half of the transform as being complex conjugates of the 1st
half
yf = 2.0/N * np.abs(pre_yf[:N//2])

#Remove DC Bias
yf[0]=0

# fig, ax = plt.subplots()
# ax.plot(xf, yf)
# ax.scatter(33, 0.006, color='red')

def find_peaks_ampd_original(x, scale=None, debug=False):
    """Find peaks in quasi-periodic noisy signals using AMPD algorithm
    Automatic Multi-Scale Peak Detection originally proposed in
    "An Efficient Algorithm for Automatic Peak Detection in
    Noisy Periodic and Quasi-Periodic Signals", Algorithms 2012, 5, 588-603
    https://doi.org/10.1109/ICRERA.2016.7884365
```

```
    Optimized implementation by Igor Gotlibovych, 2018
    Parameters
    ----------
    x : ndarray
        1-D array on which to find peaks
    scale : int, optional
        specify maximum scale window size of (2 * scale + 1)
    debug : bool, optional
        if set to True, the Local Scalogram Matrix, `LSM`, and scale with most local
maxima, `l`,
        are returned together with peak locations
    Returns
    -------
    pks: ndarray
        The ordered array of peak indices found in `x`
    """
    x = detrend(x)
    N = len(x)
    L = N // 2
    if scale:
        L = min(scale, L)

    # create LSM matix
    LSM = np.zeros((L, N), dtype=bool)
    for k in np.arange(1, L):
        LSM[k-1, k:N-k] = (x[0:N-2*k] < x[k:N-k]) & (x[k:N-k] > x[2*k:N])

    # Find scale with most maxima
    G = LSM.sum(axis=1)
    l = np.argmax(G)

    # find peaks that persist on all scales up to l
    pks_logical = np.min(LSM[0:l, :], axis=0)
    pks = np.nonzero(pks_logical)
    if debug:
        return pks, LSM, l
    return pks


def find_peaks_ampd(x, scale=None, debug=False):
    """Find peaks in quasi-periodic noisy signals using AMPD algorithm
    Extended implementation handles peaks near start/end of the signal.
    Optimized implementation by Igor Gotlibovych, 2018
    Parameters
    ----------
```

```
    x : ndarray
        1-D array on which to find peaks
    scale : int, optional
        specify maximum scale window size of (2 * scale + 1)
    debug : bool, optional
        if set to True, the Local Scalogram Matrix, `LSM`, weigted number of maxima,
'G',
        and scale at which G is maximized, `l`, are returned together with peak
locations
    Returns
    -------
    pks: ndarray
        The ordered array of peak indices found in `x`
    """
    x = detrend(x)
    N = len(x)
    L = N // 2
    if scale:
        L = min(scale, L)

    # create LSM matix
    LSM = np.ones((L, N), dtype=bool)
    for k in np.arange(1, L+1):
        LSM[k-1, 0:N-k] &= (x[0:N-k] > x[k:N])  # compare to right neighbours
        LSM[k-1, k:N] &= (x[k:N] > x[0:N-k])  # compare to left neighbours

    # Find scale with most maxima
    G = LSM.sum(axis=1)
    G = G * np.arange(N//2, N//2-L, -1)  # normalize to adjust for new edge regions
    l = np.argmax(G)

    # find peaks that persist on all scales up to l
    pks_logical = np.min(LSM[0:l, :], axis=0)
    pks = np.nonzero(pks_logical)
    if debug:
        return pks, LSM, G, l
    return pks


def find_peaks_ass_ampd(x, window=None, debug=False):
    """Find peaks in quasi-periodic noisy signals using ASS-AMPD algorithm
    Adaptive Scale Selection Automatic Multi-Scale Peak Detection, an extension of
AMPD -
    "An Efficient Algorithm for Automatic Peak Detection in
    Noisy Periodic and Quasi-Periodic Signals", Algorithms 2012, 5, 588-603
```

```
    https://doi.org/10.1109/ICRERA.2016.7884365
    Optimized implementation by Igor Gotlibovych, 2018
    Parameters
    ----------
    x : ndarray
        1-D array on which to find peaks
    window : int, optional
        sliding window size for adaptive scale selection
    debug : bool, optional
        if set to True, the Local Scalogram Matrix, `LSM`, and `adaptive_scale`,
        are returned together with peak locations
    Returns
    -------
    pks: ndarray
        The ordered array of peak indices found in `x`
    """
    x = detrend(x)
    N = len(x)
    if not window:
        window = N
    if window > N:
        window = N
    L = window // 2

    # create LSM matix
    LSM = np.ones((L, N), dtype=bool)
    for k in np.arange(1, L+1):
        LSM[k-1, 0:N-k] &= (x[0:N-k] > x[k:N])  # compare to right neighbours
        LSM[k-1, k:N] &= (x[k:N] > x[0:N-k])  # compare to left neighbours

    # Create continuos adaptive LSM
    ass_LSM = uniform_filter1d(LSM * window, window, axis=1, mode='nearest')
    normalization = np.arange(L, 0, -1)  # scale normalization weight
    ass_LSM = ass_LSM * normalization.reshape(-1, 1)

    # Find adaptive scale at each point
    adaptive_scale = ass_LSM.argmax(axis=0)

    # construct reduced LSM
    LSM_reduced = LSM[:adaptive_scale.max(), :]
    mask = (np.indices(LSM_reduced.shape)[0] > adaptive_scale) # these elements are
outside scale of interest
    LSM_reduced[mask] = 1

    # find peaks that persist on all scales up to l
```

```
    pks_logical = np.min(LSM_reduced, axis=0)
    pks = np.nonzero(pks_logical)
    if debug:
        return pks, ass_LSM, adaptive_scale
    return pks

"""Detect peaks in data based on their amplitude and other features."""


__author__ = "Marcos Duarte, https://github.com/demotu/BMC"
__version__ = "1.0.5"
__license__ = "MIT"


def detect_peaks(x, mph=None, mpd=1, threshold=0, edge='rising',
                 kpsh=False, valley=False, show=False, ax=None):

    """Detect peaks in data based on their amplitude and other features.
    Parameters
    ----------
    x : 1D array_like
        data.
    mph : {None, number}, optional (default = None)
        detect peaks that are greater than minimum peak height (if parameter
        `valley` is False) or peaks that are smaller than maximum peak height
         (if parameter `valley` is True).
    mpd : positive integer, optional (default = 1)
        detect peaks that are at least separated by minimum peak distance (in
        number of data).
    threshold : positive number, optional (default = 0)
        detect peaks (valleys) that are greater (smaller) than `threshold`
        in relation to their immediate neighbors.
    edge : {None, 'rising', 'falling', 'both'}, optional (default = 'rising')
        for a flat peak, keep only the rising edge ('rising'), only the
        falling edge ('falling'), both edges ('both'), or don't detect a
        flat peak (None).
    kpsh : bool, optional (default = False)
        keep peaks with same height even if they are closer than `mpd`.
    valley : bool, optional (default = False)
        if True (1), detect valleys (local minima) instead of peaks.
    show : bool, optional (default = False)
        if True (1), plot data in matplotlib figure.
    ax : a matplotlib.axes.Axes instance, optional (default = None).
    Returns
    -------
```

```
    ind : 1D array_like
        indeces of the peaks in `x`.
    Notes
    -----
    The detection of valleys instead of peaks is performed internally by simply
    negating the data: `ind_valleys = detect_peaks(-x)`

    The function can handle NaN's
    See this IPython Notebook [1]_.
    References
    ----------
    .. [1]
http://nbviewer.ipython.org/github/demotu/BMC/blob/master/notebooks/DetectPeaks.ipynb
    Examples
    --------
    >>> from detect_peaks import detect_peaks
    >>> x = np.random.randn(100)
    >>> x[60:81] = np.nan
    >>> # detect all peaks and plot data
    >>> ind = detect_peaks(x, show=True)
    >>> print(ind)
    >>> x = np.sin(2*np.pi*5*np.linspace(0, 1, 200)) + np.random.randn(200)/5
    >>> # set minimum peak height = 0 and minimum peak distance = 20
    >>> detect_peaks(x, mph=0, mpd=20, show=True)
    >>> x = [0, 1, 0, 2, 0, 3, 0, 2, 0, 1, 0]
    >>> # set minimum peak distance = 2
    >>> detect_peaks(x, mpd=2, show=True)
    >>> x = np.sin(2*np.pi*5*np.linspace(0, 1, 200)) + np.random.randn(200)/5
    >>> # detection of valleys instead of peaks
    >>> detect_peaks(x, mph=-1.2, mpd=20, valley=True, show=True)
    >>> x = [0, 1, 1, 0, 1, 1, 0]
    >>> # detect both edges
    >>> detect_peaks(x, edge='both', show=True)
    >>> x = [-2, 1, -2, 2, 1, 1, 3, 0]
    >>> # set threshold = 2
    >>> detect_peaks(x, threshold = 2, show=True)
    Version history
    ---------------
    '1.0.5':
        The sign of `mph` is inverted if parameter `valley` is True

    """

    x = np.atleast_1d(x).astype('float64')
    if x.size < 3:
```

```
        return np.array([], dtype=int)
    if valley:
        x = -x
        if mph is not None:
            mph = -mph
    # find indices of all peaks
    dx = x[1:] - x[:-1]
    # handle NaN's
    indnan = np.where(np.isnan(x))[0]
    if indnan.size:
        x[indnan] = np.inf
        dx[np.where(np.isnan(dx))[0]] = np.inf
    ine, ire, ife = np.array([[], [], []], dtype=int)
    if not edge:
        ine = np.where((np.hstack((dx, 0)) < 0) & (np.hstack((0, dx)) > 0))[0]
    else:
        if edge.lower() in ['rising', 'both']:
            ire = np.where((np.hstack((dx, 0)) <= 0) & (np.hstack((0, dx)) > 0))[0]
        if edge.lower() in ['falling', 'both']:
            ife = np.where((np.hstack((dx, 0)) < 0) & (np.hstack((0, dx)) >= 0))[0]
    ind = np.unique(np.hstack((ine, ire, ife)))
    # handle NaN's
    if ind.size and indnan.size:
        # NaN's and values close to NaN's cannot be peaks
        ind = ind[np.in1d(ind, np.unique(np.hstack((indnan, indnan-1, indnan+1))),
invert=True)]
    # first and last values of x cannot be peaks
    if ind.size and ind[0] == 0:
        ind = ind[1:]
    if ind.size and ind[-1] == x.size-1:
        ind = ind[:-1]
    # remove peaks < minimum peak height
    if ind.size and mph is not None:
        ind = ind[x[ind] >= mph]
    # remove peaks - neighbors < threshold
    if ind.size and threshold > 0:
        dx = np.min(np.vstack([x[ind]-x[ind-1], x[ind]-x[ind+1]]), axis=0)
        ind = np.delete(ind, np.where(dx < threshold)[0])
    # detect small peaks closer than minimum peak distance
    if ind.size and mpd > 1:
        ind = ind[np.argsort(x[ind])][::-1]  # sort ind by peak height
        idel = np.zeros(ind.size, dtype=bool)
        for i in range(ind.size):
            if not idel[i]:
                # keep peaks with the same height if kpsh is True
```

```
                idel = idel | (ind >= ind[i] - mpd) & (ind <= ind[i] + mpd) \
                    & (x[ind[i]] > x[ind] if kpsh else True)
                idel[i] = 0  # Keep current peak
        # remove the small peaks and sort back the indices by their occurrence
        ind = np.sort(ind[~idel])

    if show:
        if indnan.size:
            x[indnan] = np.nan
        if valley:
            x = -x
            if mph is not None:
                mph = -mph
        _plot(x, mph, mpd, threshold, edge, valley, ax, ind)

    return ind


def _plot(x, mph, mpd, threshold, edge, valley, ax, ind):
    """Plot results of the detect_peaks function, see its help."""
    try:
        import matplotlib.pyplot as plt
    except ImportError:
        print('matplotlib is not available.')
    else:
        if ax is None:
            _, ax = plt.subplots(1, 1, figsize=(8, 4))

        ax.plot(x, 'b', lw=1)
        if ind.size:
            label = 'valley' if valley else 'peak'
            label = label + 's' if ind.size > 1 else label
            ax.plot(ind, x[ind], '+', mfc=None, mec='r', mew=2, ms=8,
                    label='%d %s' % (ind.size, label))
            ax.legend(loc='best', framealpha=.5, numpoints=1)
        ax.set_xlim(-.02*x.size, x.size*1.02-1)
        ymin, ymax = x[np.isfinite(x)].min(), x[np.isfinite(x)].max()
        yrange = ymax - ymin if ymax > ymin else 1
        ax.set_ylim(ymin - 0.1*yrange, ymax + 0.1*yrange)
        ax.set_xlabel('Data #', fontsize=14)
        ax.set_ylabel('Amplitude', fontsize=14)
        mode = 'Valley detection' if valley else 'Peak detection'
        ax.set_title("%s (mph=%s, mpd=%d, threshold=%s, edge='%s')"
                     % (mode, str(mph), mpd, str(threshold), edge))
        # plt.grid()
```

```
        plt.show()



def scale(val, src, dst):
    """
    Scale the given value from the scale of src to the scale of dst.
    """
    return ((val - src[0]) / (src[1]-src[0])) * (dst[1]-dst[0]) + dst[0]

##################################################################
IDX_FREQ_I = 0
IDX_TIME_J = 1

##################################################################
# Sampling rate, related to the Nyquist conditions, which affects
# the range frequencies we can detect.
DEFAULT_FS = 97656

##################################################################
# Size of the FFT window, affects frequency granularity
DEFAULT_WINDOW_SIZE = 256

##################################################################
# Ratio by which each sequential window overlaps the last and the
# next window. Higher overlap will allow a higher granularity of offset
# matching, but potentially more fingerprints.
DEFAULT_OVERLAP_RATIO = 0.5

##################################################################
# Degree to which a fingerprint can be paired with its neighbors --
# higher will cause more fingerprints, but potentially better accuracy.
DEFAULT_FAN_VALUE = 20

##################################################################
# Minimum amplitude in spectrogram in order to be considered a peak.
# This can be raised to reduce number of fingerprints, but can negatively
# affect accuracy.
DEFAULT_AMP_MIN = 0

##################################################################
# Number of cells around an amplitude peak in the spectrogram in order
# for Dejavu to consider it a spectral peak. Higher values mean less
# fingerprints and faster matching, but can potentially affect accuracy.
PEAK_NEIGHBORHOOD_SIZE = 5
```

```
########################################################################
# Thresholds on how close or far fingerprints can be in time in order
# to be paired as a fingerprint. If your max is too low, higher values of
# DEFAULT_FAN_VALUE may not perform as expected.
MIN_HASH_TIME_DELTA = 0
MAX_HASH_TIME_DELTA = 200

########################################################################
# If True, will sort peaks temporally for fingerprinting;
# not sorting will cut down number of fingerprints, but potentially
# affect performance.
PEAK_SORT = True

########################################################################
# Number of bits to throw away from the front of the SHA1 hash in the
# fingerprint calculation. The more you throw away, the less storage, but
# potentially higher collisions and misclassifications when identifying songs.
FINGERPRINT_REDUCTION = 20


def fingerprint(channel_samples, Fs=DEFAULT_FS,
                wsize=DEFAULT_WINDOW_SIZE,
                wratio=DEFAULT_OVERLAP_RATIO,
                fan_value=DEFAULT_FAN_VALUE,
                    amp_min=DEFAULT_AMP_MIN):
    """
        FFT the channel, log transform output, find local maxima, then return
        locally sensitive hashes.
        """
    # Fs=DEFAULT_FS
    # wsize=DEFAULT_WINDOW_SIZE
    # wratio=DEFAULT_OVERLAP_RATIO
    # fan_value=DEFAULT_FAN_VALUE
    # amp_min=DEFAULT_AMP_MIN
    x = channel_samples

    # FFT the signal and extract frequency components
    arr2D = mlab.specgram(
        channel_samples,
        NFFT=wsize,
        Fs=Fs,
        window=mlab.window_hanning,
        noverlap=int(wsize * wratio))[0]
```

```
    # apply log transform since specgram() returns linear array
    #arr2D = 10 * np.log10(arr2D)
    #arr2D[arr2D == -np.inf] = 0  # replace infs with zeros

    # find local maxima
    #local_maxima = get_2D_peaks(arr2D, plot=False, amp_min=amp_min)

    #return local_maxima
    #return generate_hashes(local_maxima, fan_value=fan_value)


    #def get_2D_peaks(arr2D, plot=False, amp_min=DEFAULT_AMP_MIN):
        #
http://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.morphology.iterate_s
tructure.html#scipy.ndimage.morphology.iterate_structure

    struct = generate_binary_structure(2, 1)
    neighborhood = iterate_structure(struct, PEAK_NEIGHBORHOOD_SIZE)

    # find local maxima using our fliter shape
    local_max = maximum_filter(arr2D, footprint=neighborhood) == arr2D
    background = (arr2D == 0)
    eroded_background = binary_erosion(background, structure=neighborhood,
                                       border_value=1)

    #local_max=local_max.astype(np.float32)
    #eroded_background=eroded_background.astype(np.float32)
    # Boolean mask of arr2D with True at peaks
    detected_peaks = local_max

    #print(get_2D_peaks(dataWork))
    #print(fingerprint(dataWork))

    # extract peaks
    amps = arr2D[(detected_peaks)]
    ampsRatio = scale(amps, (min(amps), max(amps)), (1, 800))

    j, i = np.where(detected_peaks)

    # filter peaks
    amps = amps.flatten()
    peaks = zip(i, j, amps)
```

```
    peaks_filtered = [x for x in peaks if x[2] > amp_min]  # freq, time, amp

    # get indices for frequency and time
    frequency_idx = [x[1] for x in peaks_filtered]
    time_idx = [x[0] for x in peaks_filtered]

    return arr2D, i, j, ampsRatio, time_idx, frequency_idx,


for firpath, dirnames, filenames in walk('/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/MechDatasets/IMS/1st_test'):
    fileList = filenames
#there are 2156 files
basedir = '/Users/justinlarocque-
villiers/Desktop/CodeFolder/FileFolder/MechDatasets/IMS/1st_test/'
baseNameForDataFrame = 'File'

dt = pd.DataFrame()

nFiles = len(fileList)
count = 0
for i in range(0,len(fileList),20):
    df_time_dump = pd.read_csv(basedir+fileList[i], sep='\t', header=None)
    dt['File' + str(i) ]=df_time_dump[df_time_dump.columns[0]]-
np.mean(df_time_dump[df_time_dump.columns[0]]) #only take the 1st column (x axis)
dt.head(10)
df = pd.read_csv('FrequencyValues.csv')
df_stats = df.describe()
df_stats
plt.plot(df_stats.iloc[1,:], color='blue')
plt.plot(df_stats.iloc[2,:], color='red')

df_fingerprep = pd.DataFrame()
#Build a dataframe with content rows: arr2D (drop), i, j, ampsRatio, time_idx (drop),
frequency_idx (drop),
for i in range(0,len(dt.columns)):
    fingerholder = fingerprint(dt.iloc[:,i])
    df_fingerprep['Sample:'+ str(i)] = fingerholder[1:4]
#df_fingerprep.drop([0,4,5], inplace=True)
df_fingerprep.head(10)

df_fingerprep_lab_good = pd.DataFrame()

#Build a dataframe with content rows: arr2D (drop), i, j, ampsRatio, time_idx (drop),
frequency_idx (drop),
for i in range(0,nFiles_lab//2):
```

```
    df_fingerprep_lab_good[str(filenames[i])] = fingerprint(dt_lab.iloc[:,i])
df_fingerprep_lab_good.drop([0,4,5], inplace=True)
df_fingerprep_lab_good.head(10)

df_fingerprep_lab_fault = pd.DataFrame()

#Build a dataframe with content rows: arr2D (drop), i, j, ampsRatio, time_idx (drop),
frequency_idx (drop),
for i in range(nFiles_lab//2, nFiles_lab):
    df_fingerprep_lab_fault[str(filenames[i])] = fingerprint(dt_lab.iloc[:,i])
df_fingerprep_lab_fault.drop([0,4,5], inplace=True)
df_fingerprep_lab_fault.head(10)


plt.plot(df_fingerprep.iloc[1,0], df_fingerprep.iloc[2,0], marker = 'x',markersize=15,
linewidth=0.1, alpha = 0.9)
plt.plot(df_fingerprep.iloc[1,-1], df_fingerprep.iloc[2,-1], marker =
'x',markersize=15, linewidth=0.1, alpha = 0.9)

for i in range(0, (len(df_fingerprep_lab_good.columns))-1):
    plt.scatter((df_fingerprep_lab_good.iloc[1,i]), df_fingerprep_lab_good.iloc[2,i],
s=10, alpha=0.5, color='blue')

for i in range(0, (len(df_fingerprep_lab_fault.columns))):
    plt.scatter(df_fingerprep_lab_fault.iloc[1,i], df_fingerprep_lab_fault.iloc[2,i],
s=10, alpha=0.5, color='red')

#Build a dataframe with content rows: arr2D (drop), i, j, ampsRatio, time_idx (drop),
frequency_idx (drop),
for i in range(0, len(df_bearing.columns)):
    df_fingerprep_hsbearing['File:'+str(i)] = fingerprint(df_bearing.iloc[0:97656,i])
df_fingerprep_hsbearing.drop([0,4,5], inplace=True)
df_fingerprep_hsbearing.head(10)
```

```
#!/usr/bin/python2


# Copyright (C) 2016 Sixten Bergman
# License WTFPL
#
# This program is free software. It comes without any warranty, to the extent
# permitted by applicable law.
# You can redistribute it and/or modify it under the terms of Public License, Version
2, as published by Sam Hocevar. See
# http://www.wtfpl.net/ for more details.
#
# note that the function peakdetect is derived from code which was released to
# public domain see: http://billauer.co.il/peakdet.html
#

import logging
from math import pi, log
import numpy as np
import pylab
from scipy import fft, ifft
from scipy.optimize import curve_fit
from scipy.signal import cspline1d_eval, cspline1d
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [30, 14]


__all__ = [
        "peakdetect",
        "peakdetect_fft",
        "peakdetect_parabola",
        "peakdetect_sine",
        "peakdetect_sine_locked",
        "peakdetect_spline",
        "peakdetect_zero_crossing",
        "zero_crossings",
        "zero_crossings_sine_fit"
        ]



def _datacheck_peakdetect(x_axis, y_axis):
    if x_axis is None:
        x_axis = range(len(y_axis))
```

```
    if len(y_axis) != len(x_axis):
        raise ValueError(
                "Input vectors y_axis and x_axis must have same length")

    #needs to be a numpy array
    y_axis = np.array(y_axis)
    x_axis = np.array(x_axis)
    return x_axis, y_axis


def _pad(fft_data, pad_len):
    """
    Pads fft data to interpolate in time domain

    keyword arguments:
    fft_data -- the fft
    pad_len --  By how many times the time resolution should be increased by

    return: padded list
    """
    l = len(fft_data)
    n = _n(l * pad_len)
    fft_data = list(fft_data)

    return fft_data[:l // 2] + [0] * (2**n-l) + fft_data[l // 2:]

def _n(x):
    """
    Find the smallest value for n, which fulfils 2**n >= x

    keyword arguments:
    x -- the value, which 2**n must surpass

    return: the integer n
    """
    return int(log(x)/log(2)) + 1


def _peakdetect_parabola_fitter(raw_peaks, x_axis, y_axis, points):
    """
    Performs the actual parabola fitting for the peakdetect_parabola function.

    keyword arguments:
    raw_peaks -- A list of either the maxima or the minima peaks, as given
        by the peakdetect functions, with index used as x-axis
```

```
    x_axis -- A numpy array of all the x values

    y_axis -- A numpy array of all the y values

    points -- How many points around the peak should be used during curve
        fitting, must be odd.


    return: A list giving all the peaks and the fitted waveform, format:
        [[x, y, [fitted_x, fitted_y]]]

    """
    func = lambda x, a, tau, c: a * ((x - tau) ** 2) + c
    fitted_peaks = []
    distance = abs(x_axis[raw_peaks[1][0]] - x_axis[raw_peaks[0][0]]) / 4
    for peak in raw_peaks:
        index = peak[0]
        x_data = x_axis[index - points // 2: index + points // 2 + 1]
        y_data = y_axis[index - points // 2: index + points // 2 + 1]
        # get a first approximation of tau (peak position in time)
        tau = x_axis[index]
        # get a first approximation of peak amplitude
        c = peak[1]
        a = np.sign(c) * (-1) * (np.sqrt(abs(c))/distance)**2
        """Derived from ABC formula to result in a solution where A=(rot(c)/t)**2"""

        # build list of approximations

        p0 = (a, tau, c)
        popt, pcov = curve_fit(func, x_data, y_data, p0)
        # retrieve tau and c i.e x and y value of peak
        x, y = popt[1:3]

        # create a high resolution data set for the fitted waveform
        x2 = np.linspace(x_data[0], x_data[-1], points * 10)
        y2 = func(x2, *popt)

        fitted_peaks.append([x, y, [x2, y2]])

    return fitted_peaks


def peakdetect_parabole(*args, **kwargs):
    """
```

```
    Misspelling of peakdetect_parabola
    function is deprecated please use peakdetect_parabola
    """
    logging.warn("peakdetect_parabole is deprecated due to misspelling use:
peakdetect_parabola")

    return peakdetect_parabola(*args, **kwargs)


def peakdetect(y_axis, x_axis = None, lookahead = 200, delta=0):
    """
    Converted from/based on a MATLAB script at:
    http://billauer.co.il/peakdet.html

    function for detecting local maxima and minima in a signal.
    Discovers peaks by searching for values which are surrounded by lower
    or larger values for maxima and minima respectively

    keyword arguments:
    y_axis -- A list containing the signal over which to find peaks

    x_axis -- A x-axis whose values correspond to the y_axis list and is used
        in the return to specify the position of the peaks. If omitted an
        index of the y_axis is used.
        (default: None)

    lookahead -- distance to look ahead from a peak candidate to determine if
        it is the actual peak
        (default: 200)
        '(samples / period) / f' where '4 >= f >= 1.25' might be a good value

    delta -- this specifies a minimum difference between a peak and
        the following points, before a peak may be considered a peak. Useful
        to hinder the function from picking up false peaks towards to end of
        the signal. To work well delta should be set to delta >= RMSnoise * 5.
        (default: 0)
            When omitted delta function causes a 20% decrease in speed.
            When used Correctly it can double the speed of the function


    return: two lists [max_peaks, min_peaks] containing the positive and
        negative peaks respectively. Each cell of the lists contains a tuple
        of: (position, peak_value)
        to get the average peak value do: np.mean(max_peaks, 0)[1] on the
        results to unpack one of the lists into x, y coordinates do:
```

```
    x, y = zip(*max_peaks)
"""
max_peaks = []
min_peaks = []
dump = []   #Used to pop the first hit which almost always is false

# check input data
x_axis, y_axis = _datacheck_peakdetect(x_axis, y_axis)
# store data length for later use
length = len(y_axis)


#perform some checks
if lookahead < 1:
    raise ValueError("Lookahead must be '1' or above in value")
if not (np.isscalar(delta) and delta >= 0):
    raise ValueError("delta must be a positive number")

#maxima and minima candidates are temporarily stored in
#mx and mn respectively
mn, mx = np.Inf, -np.Inf

#Only detect peak if there is 'lookahead' amount of points after it
for index, (x, y) in enumerate(zip(x_axis[:-lookahead],
                                   y_axis[:-lookahead])):
    if y > mx:
        mx = y
        mxpos = x
    if y < mn:
        mn = y
        mnpos = x

    ####look for max####
    if y < mx-delta and mx != np.Inf:
        #Maxima peak candidate found
        #look ahead in signal to ensure that this is a peak and not jitter
        if y_axis[index:index+lookahead].max() < mx:
            max_peaks.append([mxpos, mx])
            dump.append(True)
            #set algorithm to only find minima now
            mx = np.Inf
            mn = np.Inf
            if index+lookahead >= length:
                #end is within lookahead no more peaks can be found
                break
```

```
                continue
            #else:  #slows shit down this does
            #    mx = ahead
            #    mxpos = x_axis[np.where(y_axis[index:index+lookahead]==mx)]

        ####look for min####
        if y > mn+delta and mn != -np.Inf:
            #Minima peak candidate found
            #look ahead in signal to ensure that this is a peak and not jitter
            if y_axis[index:index+lookahead].min() > mn:
                min_peaks.append([mnpos, mn])
                dump.append(False)
                #set algorithm to only find maxima now
                mn = -np.Inf
                mx = -np.Inf
                if index+lookahead >= length:
                    #end is within lookahead no more peaks can be found
                    break
            #else:  #slows shit down this does
            #    mn = ahead
            #    mnpos = x_axis[np.where(y_axis[index:index+lookahead]==mn)]


    #Remove the false hit on the first value of the y_axis
    try:
        if dump[0]:
            max_peaks.pop(0)
        else:
            min_peaks.pop(0)
        del dump
    except IndexError:
        #no peaks were found, should the function return empty lists?
        pass

    return [max_peaks, min_peaks]


def peakdetect_fft(y_axis, x_axis, pad_len = 20):
    """
    Performs a FFT calculation on the data and zero-pads the results to
    increase the time domain resolution after performing the inverse fft and
    send the data to the 'peakdetect' function for peak
    detection.

    Omitting the x_axis is forbidden as it would make the resulting x_axis
```

```
value silly if it was returned as the index 50.234 or similar.

Will find at least 1 less peak then the 'peakdetect_zero_crossing'
function, but should result in a more precise value of the peak as
resolution has been increased. Some peaks are lost in an attempt to
minimize spectral leakage by calculating the fft between two zero
crossings for n amount of signal periods.

The biggest time eater in this function is the ifft and thereafter it's
the 'peakdetect' function which takes only half the time of the ifft.
Speed improvements could include to check if 2**n points could be used for
fft and ifft or change the 'peakdetect' to the 'peakdetect_zero_crossing',
which is maybe 10 times faster than 'peakdetct'. The pro of 'peakdetect'
is that it results in one less lost peak. It should also be noted that the
time used by the ifft function can change greatly depending on the input.

keyword arguments:
y_axis -- A list containing the signal over which to find peaks

x_axis -- A x-axis whose values correspond to the y_axis list and is used
    in the return to specify the position of the peaks.

pad_len -- By how many times the time resolution should be
    increased by, e.g. 1 doubles the resolution. The amount is rounded up
    to the nearest 2**n amount
    (default: 20)


return: two lists [max_peaks, min_peaks] containing the positive and
    negative peaks respectively. Each cell of the lists contains a tuple
    of: (position, peak_value)
    to get the average peak value do: np.mean(max_peaks, 0)[1] on the
    results to unpack one of the lists into x, y coordinates do:
    x, y = zip(*max_peaks)
"""
# check input data
x_axis, y_axis = _datacheck_peakdetect(x_axis, y_axis)
zero_indices = zero_crossings(y_axis, window_len = 11)
#select a n amount of periods
last_indice = - 1 - (1 - len(zero_indices) & 1)
###
# Calculate the fft between the first and last zero crossing
# this method could be ignored if the beginning and the end of the signal
# are unnecessary as any errors induced from not using whole periods
# should mainly manifest in the beginning and the end of the signal, but
```

```
    # not in the rest of the signal
    # this is also unnecessary if the given data is an amount of whole periods
    ###
    fft_data = fft(y_axis[zero_indices[0]:zero_indices[last_indice]])
    padd = lambda x, c: x[:len(x) // 2] + [0] * c + x[len(x) // 2:]
    n = lambda x: int(log(x)/log(2)) + 1
    # pads to 2**n amount of samples
    fft_padded = padd(list(fft_data), 2 **
                n(len(fft_data) * pad_len) - len(fft_data))

    # There is amplitude decrease directly proportional to the sample increase
    sf = len(fft_padded) / float(len(fft_data))
    # There might be a leakage giving the result an imaginary component
    # Return only the real component
    y_axis_ifft = ifft(fft_padded).real * sf #(pad_len + 1)
    x_axis_ifft = np.linspace(
                x_axis[zero_indices[0]], x_axis[zero_indices[last_indice]],
                len(y_axis_ifft))
    # get the peaks to the interpolated waveform
    max_peaks, min_peaks = peakdetect(y_axis_ifft, x_axis_ifft, 500,
                                      delta = abs(np.diff(y_axis).max() * 2))
    #max_peaks, min_peaks = peakdetect_zero_crossing(y_axis_ifft, x_axis_ifft)

    # store one 20th of a period as waveform data
    data_len = int(np.diff(zero_indices).mean()) / 10
    data_len += 1 - data_len & 1


    return [max_peaks, min_peaks]


def peakdetect_parabola(y_axis, x_axis, points = 31):
    """
    Function for detecting local maxima and minima in a signal.
    Discovers peaks by fitting the model function: y = k (x - tau) ** 2 + m
    to the peaks. The amount of points used in the fitting is set by the
    points argument.

    Omitting the x_axis is forbidden as it would make the resulting x_axis
    value silly, if it was returned as index 50.234 or similar.

    will find the same amount of peaks as the 'peakdetect_zero_crossing'
    function, but might result in a more precise value of the peak.

    keyword arguments:
```

```
    y_axis -- A list containing the signal over which to find peaks

    x_axis -- A x-axis whose values correspond to the y_axis list and is used
        in the return to specify the position of the peaks.

    points -- How many points around the peak should be used during curve
        fitting (default: 31)


    return: two lists [max_peaks, min_peaks] containing the positive and
        negative peaks respectively. Each cell of the lists contains a tuple
        of: (position, peak_value)
        to get the average peak value do: np.mean(max_peaks, 0)[1] on the
        results to unpack one of the lists into x, y coordinates do:
        x, y = zip(*max_peaks)
    """
    # check input data
    x_axis, y_axis = _datacheck_peakdetect(x_axis, y_axis)
    # make the points argument odd
    points += 1 - points % 2
    #points += 1 - int(points) & 1 slower when int conversion needed

    # get raw peaks
    max_raw, min_raw = peakdetect_zero_crossing(y_axis)

    # define output variable
    max_peaks = []
    min_peaks = []

    max_ = _peakdetect_parabola_fitter(max_raw, x_axis, y_axis, points)
    min_ = _peakdetect_parabola_fitter(min_raw, x_axis, y_axis, points)

    max_peaks = map(lambda x: [x[0], x[1]], max_)
    max_fitted = map(lambda x: x[-1], max_)
    min_peaks = map(lambda x: [x[0], x[1]], min_)
    min_fitted = map(lambda x: x[-1], min_)

    return [max_peaks, min_peaks]


def peakdetect_sine(y_axis, x_axis, points = 31, lock_frequency = False):
    """
    Function for detecting local maxima and minima in a signal.
    Discovers peaks by fitting the model function:
    y = A * sin(2 * pi * f * (x - tau)) to the peaks. The amount of points used
```

```
in the fitting is set by the points argument.

Omitting the x_axis is forbidden as it would make the resulting x_axis
value silly if it was returned as index 50.234 or similar.

will find the same amount of peaks as the 'peakdetect_zero_crossing'
function, but might result in a more precise value of the peak.

The function might have some problems if the sine wave has a
non-negligible total angle i.e. a k*x component, as this messes with the
internal offset calculation of the peaks, might be fixed by fitting a
y = k * x + m function to the peaks for offset calculation.

keyword arguments:
y_axis -- A list containing the signal over which to find peaks

x_axis -- A x-axis whose values correspond to the y_axis list and is used
    in the return to specify the position of the peaks.

points -- How many points around the peak should be used during curve
    fitting (default: 31)

lock_frequency -- Specifies if the frequency argument of the model
    function should be locked to the value calculated from the raw peaks
    or if optimization process may tinker with it.
    (default: False)


return: two lists [max_peaks, min_peaks] containing the positive and
    negative peaks respectively. Each cell of the lists contains a tuple
    of: (position, peak_value)
    to get the average peak value do: np.mean(max_peaks, 0)[1] on the
    results to unpack one of the lists into x, y coordinates do:
    x, y = zip(*max_peaks)
"""
# check input data
x_axis, y_axis = _datacheck_peakdetect(x_axis, y_axis)
# make the points argument odd
points += 1 - points % 2
#points += 1 - int(points) & 1 slower when int conversion needed

# get raw peaks
max_raw, min_raw = peakdetect_zero_crossing(y_axis)

# define output variable
```

```
max_peaks = []
min_peaks = []

# get global offset
offset = np.mean([np.mean(max_raw, 0)[1], np.mean(min_raw, 0)[1]])
# fitting a k * x + m function to the peaks might be better
#offset_func = lambda x, k, m: k * x + m

# calculate an approximate frequency of the signal
Hz_h_peak = np.diff(zip(*max_raw)[0]).mean()
Hz_l_peak = np.diff(zip(*min_raw)[0]).mean()
Hz = 1 / np.mean([Hz_h_peak, Hz_l_peak])



# model function
# if cosine is used then tau could equal the x position of the peak
# if sine were to be used then tau would be the first zero crossing
if lock_frequency:
    func = lambda x_ax, A, tau: A * np.sin(
        2 * pi * Hz * (x_ax - tau) + pi / 2)
else:
    func = lambda x_ax, A, Hz, tau: A * np.sin(
        2 * pi * Hz * (x_ax - tau) + pi / 2)
#func = lambda x_ax, A, Hz, tau: A * np.cos(2 * pi * Hz * (x_ax - tau))


#get peaks
fitted_peaks = []
for raw_peaks in [max_raw, min_raw]:
    peak_data = []
    for peak in raw_peaks:
        index = peak[0]
        x_data = x_axis[index - points // 2: index + points // 2 + 1]
        y_data = y_axis[index - points // 2: index + points // 2 + 1]
        # get a first approximation of tau (peak position in time)
        tau = x_axis[index]
        # get a first approximation of peak amplitude
        A = peak[1]

        # build list of approximations
        if lock_frequency:
            p0 = (A, tau)
        else:
            p0 = (A, Hz, tau)
```

```
            # subtract offset from wave-shape
            y_data -= offset
            popt, pcov = curve_fit(func, x_data, y_data, p0)
            # retrieve tau and A i.e x and y value of peak
            x = popt[-1]
            y = popt[0]

            # create a high resolution data set for the fitted waveform
            x2 = np.linspace(x_data[0], x_data[-1], points * 10)
            y2 = func(x2, *popt)

            # add the offset to the results
            y += offset
            y2 += offset
            y_data += offset

            peak_data.append([x, y, [x2, y2]])

        fitted_peaks.append(peak_data)

    # structure date for output
    max_peaks = map(lambda x: [x[0], x[1]], fitted_peaks[0])
    max_fitted = map(lambda x: x[-1], fitted_peaks[0])
    min_peaks = map(lambda x: [x[0], x[1]], fitted_peaks[1])
    min_fitted = map(lambda x: x[-1], fitted_peaks[1])


    return [max_peaks, min_peaks]


def peakdetect_sine_locked(y_axis, x_axis, points = 31):
    """
    Convenience function for calling the 'peakdetect_sine' function with
    the lock_frequency argument as True.

    keyword arguments:
    y_axis -- A list containing the signal over which to find peaks
    x_axis -- A x-axis whose values correspond to the y_axis list and is used
        in the return to specify the position of the peaks.
    points -- How many points around the peak should be used during curve
        fitting (default: 31)

    return: see the function 'peakdetect_sine'
    """
```

```
    return peakdetect_sine(y_axis, x_axis, points, True)


def peakdetect_spline(y_axis, x_axis, pad_len=20):
    """
    Performs a b-spline interpolation on the data to increase resolution and
    send the data to the 'peakdetect_zero_crossing' function for peak
    detection.

    Omitting the x_axis is forbidden as it would make the resulting x_axis
    value silly if it was returned as the index 50.234 or similar.

    will find the same amount of peaks as the 'peakdetect_zero_crossing'
    function, but might result in a more precise value of the peak.

    keyword arguments:
    y_axis -- A list containing the signal over which to find peaks

    x_axis -- A x-axis whose values correspond to the y_axis list and is used
        in the return to specify the position of the peaks.
        x-axis must be equally spaced.

    pad_len -- By how many times the time resolution should be increased by,
        e.g. 1 doubles the resolution.
        (default: 20)


    return: two lists [max_peaks, min_peaks] containing the positive and
        negative peaks respectively. Each cell of the lists contains a tuple
        of: (position, peak_value)
        to get the average peak value do: np.mean(max_peaks, 0)[1] on the
        results to unpack one of the lists into x, y coordinates do:
        x, y = zip(*max_peaks)
    """
    # check input data
    x_axis, y_axis = _datacheck_peakdetect(x_axis, y_axis)
    # could perform a check if x_axis is equally spaced
    #if np.std(np.diff(x_axis)) > 1e-15: raise ValueError
    # perform spline interpolations
    dx = x_axis[1] - x_axis[0]
    x_interpolated = np.linspace(x_axis.min(), x_axis.max(), len(x_axis) * (pad_len +
1))
    cj = cspline1d(y_axis)
    y_interpolated = cspline1d_eval(cj, x_interpolated, dx=dx,x0=x_axis[0])
    # get peaks
```

```
    max_peaks, min_peaks = peakdetect_zero_crossing(y_interpolated, x_interpolated)

    return [max_peaks, min_peaks]

def peakdetect_zero_crossing(y_axis, x_axis = None, window = 11):
    """
    Function for detecting local maxima and minima in a signal.
    Discovers peaks by dividing the signal into bins and retrieving the
    maximum and minimum value of each the even and odd bins respectively.
    Division into bins is performed by smoothing the curve and finding the
    zero crossings.

    Suitable for repeatable signals, where some noise is tolerated. Executes
    faster than 'peakdetect', although this function will break if the offset
    of the signal is too large. It should also be noted that the first and
    last peak will probably not be found, as this function only can find peaks
    between the first and last zero crossing.

    keyword arguments:
    y_axis -- A list containing the signal over which to find peaks

    x_axis -- A x-axis whose values correspond to the y_axis list
        and is used in the return to specify the position of the peaks. If
        omitted an index of the y_axis is used.
        (default: None)

    window -- the dimension of the smoothing window; should be an odd integer
        (default: 11)


    return: two lists [max_peaks, min_peaks] containing the positive and
        negative peaks respectively. Each cell of the lists contains a tuple
        of: (position, peak_value)
        to get the average peak value do: np.mean(max_peaks, 0)[1] on the
        results to unpack one of the lists into x, y coordinates do:
        x, y = zip(*max_peaks)
    """
    # check input data
    x_axis, y_axis = _datacheck_peakdetect(x_axis, y_axis)

    zero_indices = zero_crossings(y_axis, window_len = window)
    period_lengths = np.diff(zero_indices)

    bins_y = [y_axis[index:index + diff] for index, diff in
        zip(zero_indices, period_lengths)]
```

```
    bins_x = [x_axis[index:index + diff] for index, diff in
        zip(zero_indices, period_lengths)]

    even_bins_y = bins_y[::2]
    odd_bins_y = bins_y[1::2]
    even_bins_x = bins_x[::2]
    odd_bins_x = bins_x[1::2]
    hi_peaks_x = []
    lo_peaks_x = []

    #check if even bin contains maxima
    if abs(even_bins_y[0].max()) > abs(even_bins_y[0].min()):
        hi_peaks = [bin.max() for bin in even_bins_y]
        lo_peaks = [bin.min() for bin in odd_bins_y]
        # get x values for peak
        for bin_x, bin_y, peak in zip(even_bins_x, even_bins_y, hi_peaks):
            hi_peaks_x.append(bin_x[np.where(bin_y==peak)[0][0]])
        for bin_x, bin_y, peak in zip(odd_bins_x, odd_bins_y, lo_peaks):
            lo_peaks_x.append(bin_x[np.where(bin_y==peak)[0][0]])
    else:
        hi_peaks = [bin.max() for bin in odd_bins_y]
        lo_peaks = [bin.min() for bin in even_bins_y]
        # get x values for peak
        for bin_x, bin_y, peak in zip(odd_bins_x, odd_bins_y, hi_peaks):
            hi_peaks_x.append(bin_x[np.where(bin_y==peak)[0][0]])
        for bin_x, bin_y, peak in zip(even_bins_x, even_bins_y, lo_peaks):
            lo_peaks_x.append(bin_x[np.where(bin_y==peak)[0][0]])

    max_peaks = [[x, y] for x,y in zip(hi_peaks_x, hi_peaks)]
    min_peaks = [[x, y] for x,y in zip(lo_peaks_x, lo_peaks)]

    return [max_peaks, min_peaks]


def _smooth(x, window_len=11, window="hanning"):
    """
    smooth the data using a window of the requested size.

    This method is based on the convolution of a scaled window on the signal.
    The signal is prepared by introducing reflected copies of the signal
    (with the window size) in both ends so that transient parts are minimized
    in the beginning and end part of the output signal.

    keyword arguments:
    x -- the input signal
```

```
window_len -- the dimension of the smoothing window; should be an odd
    integer (default: 11)

window -- the type of window from 'flat', 'hanning', 'hamming',
    'bartlett', 'blackman', where flat is a moving average
    (default: 'hanning')

return: the smoothed signal

example:
t = linspace(-2,2,0.1)
x = sin(t)+randn(len(t))*0.1
y = _smooth(x)

see also:

numpy.hanning, numpy.hamming, numpy.bartlett, numpy.blackman,
numpy.convolve, scipy.signal.lfilter
"""
if x.ndim != 1:
    raise ValueError("smooth only accepts 1 dimension arrays.")

if x.size < window_len:
    raise ValueError("Input vector needs to be bigger than window size.")

if window_len<3:
    return x
#declare valid windows in a dictionary
window_funcs = {
    "flat": lambda _len: np.ones(_len, "d"),
    "hanning": np.hanning,
    "hamming": np.hamming,
    "bartlett": np.bartlett,
    "blackman": np.blackman
    }

s = np.r_[x[window_len-1:0:-1], x, x[-1:-window_len:-1]]
try:
    w = window_funcs[window](window_len)
except KeyError:
    raise ValueError(
        "Window is not one of '{0}', '{1}', '{2}', '{3}', '{4}'".format(
        *window_funcs.keys()))
```

```
    y = np.convolve(w / w.sum(), s, mode = "valid")

    return y


def zero_crossings(y_axis, window_len = 11,
    window_f="hanning", offset_corrected=False):
    """
    Algorithm to find zero crossings. Smooths the curve and finds the
    zero-crossings by looking for a sign change.


    keyword arguments:
    y_axis -- A list containing the signal over which to find zero-crossings

    window_len -- the dimension of the smoothing window; should be an odd
        integer (default: 11)

    window_f -- the type of window from 'flat', 'hanning', 'hamming',
        'bartlett', 'blackman' (default: 'hanning')

    offset_corrected -- Used for recursive calling to remove offset when needed


    return: the index for each zero-crossing
    """
    # smooth the curve
    length = len(y_axis)

    # discard tail of smoothed signal
    y_axis = _smooth(y_axis, window_len, window_f)[:length]
    indices = np.where(np.diff(np.sign(y_axis)))[0]

    # check if zero-crossings are valid
    diff = np.diff(indices)
    if diff.std() / diff.mean() > 0.1:
        #Possibly bad zero crossing, see if it's offsets
        if ((diff[::2].std() / diff[::2].mean()) < 0.1 and
        (diff[1::2].std() / diff[1::2].mean()) < 0.1 and
        not offset_corrected):
            #offset present attempt to correct by subtracting the average
            offset = np.mean([y_axis.max(), y_axis.min()])
            return zero_crossings(y_axis-offset, window_len, window_f, True)
        #Invalid zero crossings and the offset has been removed
        print(diff.std() / diff.mean())
```

```
        print(np.diff(indices))
        raise ValueError(
            "False zero-crossings found, indicates problem {0!s} or {1!s}".format(
            "with smoothing window", "unhandled problem with offset"))
    # check if any zero crossings were found
    if len(indices) < 1:
        raise ValueError("No zero crossings found")
    #remove offset from indices due to filter function when returning
    return indices - (window_len // 2 - 1)
    # used this to test the fft function's sensitivity to spectral leakage
    #return indices + np.asarray(30 * np.random.randn(len(indices)), int)

############################Frequency calculation#################################
#    diff = np.diff(indices)
#    time_p_period = diff.mean()
#
#    if diff.std() / time_p_period > 0.1:
#        raise ValueError(
#            "smoothing window too small, false zero-crossing found")
#
#    #return frequency
#    return 1.0 / time_p_period
##################################################################################


def zero_crossings_sine_fit(y_axis, x_axis, fit_window = None, smooth_window = 11):
    """
    Detects the zero crossings of a signal by fitting a sine model function
    around the zero crossings:
    y = A * sin(2 * pi * Hz * (x - tau)) + k * x + m
    Only tau (the zero crossing) is varied during fitting.

    Offset and a linear drift of offset is accounted for by fitting a linear
    function the negative respective positive raw peaks of the wave-shape and
    the amplitude is calculated using data from the offset calculation i.e.
    the 'm' constant from the negative peaks is subtracted from the positive
    one to obtain amplitude.

    Frequency is calculated using the mean time between raw peaks.

    Algorithm seems to be sensitive to first guess e.g. a large smooth_window
    will give an error in the results.


    keyword arguments:
```

```
y_axis -- A list containing the signal over which to find peaks

x_axis -- A x-axis whose values correspond to the y_axis list
    and is used in the return to specify the position of the peaks. If
    omitted an index of the y_axis is used. (default: None)

fit_window -- Number of points around the approximate zero crossing that
    should be used when fitting the sine wave. Must be small enough that
    no other zero crossing will be seen. If set to none then the mean
    distance between zero crossings will be used (default: None)

smooth_window -- the dimension of the smoothing window; should be an odd
    integer (default: 11)


return: A list containing the positions of all the zero crossings.
"""
# check input data
x_axis, y_axis = _datacheck_peakdetect(x_axis, y_axis)
#get first guess
zero_indices = zero_crossings(y_axis, window_len = smooth_window)
#modify fit_window to show distance per direction
if fit_window == None:
    fit_window = np.diff(zero_indices).mean() // 3
else:
    fit_window = fit_window // 2

#x_axis is a np array, use the indices to get a subset with zero crossings
approx_crossings = x_axis[zero_indices]


#get raw peaks for calculation of offsets and frequency
raw_peaks = peakdetect_zero_crossing(y_axis, x_axis)
#Use mean time between peaks for frequency
ext = lambda x: list(zip(*x)[0])
_diff = map(np.diff, map(ext, raw_peaks))


Hz = 1 / np.mean(map(np.mean, _diff))
#Hz = 1 / np.diff(approx_crossings).mean() #probably bad precision


#offset model function
offset_func = lambda x, k, m: k * x + m
k = []
```

```
    m = []
    amplitude = []

    for peaks in raw_peaks:
        #get peak data as nparray
        x_data, y_data = map(np.asarray, zip(*peaks))
        #x_data = np.asarray(x_data)
        #y_data = np.asarray(y_data)
        #calc first guess
        A = np.mean(y_data)
        p0 = (0, A)
        popt, pcov = curve_fit(offset_func, x_data, y_data, p0)
        #append results
        k.append(popt[0])
        m.append(popt[1])
        amplitude.append(abs(A))

    #store offset constants
    p_offset = (np.mean(k), np.mean(m))
    A = m[0] - m[1]
    #define model function to fit to zero crossing
    #y = A * sin(2*pi * Hz * (x - tau)) + k * x + m
    func = lambda x, tau: A * np.sin(2 * pi * Hz * (x - tau)) + offset_func(x,
*p_offset)


    #get true crossings
    true_crossings = []
    for indice, crossing in zip(zero_indices, approx_crossings):
        p0 = (crossing, )
        subset_start = max(indice - fit_window, 0.0)
        subset_end = min(indice + fit_window + 1, len(x_axis) - 1.0)
        x_subset = np.asarray(x_axis[subset_start:subset_end])
        y_subset = np.asarray(y_axis[subset_start:subset_end])
        #fit
        popt, pcov = curve_fit(func, x_subset, y_subset, p0)

        true_crossings.append(popt[0])


    return true_crossings
```

```
def _test_zero():
    _max, _min = peakdetect_zero_crossing(y,x)
def _test():
    _max, _min = peakdetect(y,x, delta=0.30)

def _test_graph():
    i = 10000
    x = np.linspace(0,3.7*pi,i)
    y = (0.3*np.sin(x) + np.sin(1.3 * x) + 0.9 * np.sin(4.2 * x) + 0.06 *
    np.random.randn(i))
    y *= -1
    x = range(i)

    _max, _min = peakdetect(y,x,750, 0.30)
    xm = [p[0] for p in _max]
    ym = [p[1] for p in _max]
    xn = [p[0] for p in _min]
    yn = [p[1] for p in _min]

    plot = pylab.plot(x,y)
    pylab.hold(True)
    pylab.plot(xm, ym, "r+")
    pylab.plot(xn, yn, "g+")

    _max, _min = peak_det_bad.peakdetect(y, 0.7, x)
    xm = [p[0] for p in _max]
    ym = [p[1] for p in _max]
    xn = [p[0] for p in _min]
    yn = [p[1] for p in _min]
    pylab.plot(xm, ym, "y*")
    pylab.plot(xn, yn, "k*")
    pylab.show()

def _test_graph_cross(window = 11):
    i = 10000
    x = np.linspace(0,8.7*pi,i)
    y = (2*np.sin(x) + 0.006 *
    np.random.randn(i))
    y *= -1
    pylab.plot(x,y)
    #pylab.show()


    crossings = zero_crossings_sine_fit(y,x, smooth_window = window)
    y_cross = [0] * len(crossings)
```

```
    plot = pylab.plot(x,y)
    pylab.hold(True)
    pylab.plot(crossings, y_cross, "b+")
    pylab.show()

if __name__ == "__main__":
    from math import pi
    import pylab

    i = 10000
    x = np.linspace(0,3.7*pi,i)
    y = (0.3*np.sin(x) + np.sin(1.3 * x) + 0.9 * np.sin(4.2 * x) + 0.06 *
    np.random.randn(i))
    y *= -1

    _max, _min = peakdetect(y, x, 750, 0.30)
    xm = [p[0] for p in _max]
    ym = [p[1] for p in _max]
    xn = [p[0] for p in _min]
    yn = [p[1] for p in _min]

    plt.plot(x, y)
    #pylab.hold(True)
    plt.scatter(xm, ym, color="r", s=200)
    plt.scatter(xn, yn, color="g", s=200)

    plt.show()
```